# Phasemeter Interface (PMI)

## Technical Reference Manual

Software Version 1.1.2

26 february 2010

by Daniel Gering

**Abstract**

The purpose of the AEI 10 m Prototype Interferometer is to test and develop several techniques for potential upgrades of the gravitational wave detector GEO600 [4] and to explore macroscopical quantum mechanical effects. In addition to the main interferometer the 10 m Prototype includes a set of three auxiliary interferometers, which form the so-called Suspension Platform interferometer (SPI). The SPI measures the relative motion between three seismically isolated optical tables, which are located at the corners of an L-shaped ultra-high vacuum system. The measured motion is used to derive a feedback signal to minimize the displacement of the tables with respect to each other. Besides the interferometer optics, the SPI consists of a phasemeter to which all quadrant photo diodes are connected to. The phasemeter converts the analog signals into digital and applies a Fourier transformation. The data is then transmitted to a realtime computer system which controls the actuators of the tables to minimize their residual motion. For data transmission, the phasemeter is equipped with an Enhanced Parallel Port (EPP). However, it is not possible to connect the realtime computer system directly to the phasemeter. The reasons are: a computer-sided parallel port cannot read the data fast enough to achieve a sufficient transfer rate; additionally, the EPP cable length constraint of a few meter is not suitable, because due to infrastructural reasons the computer system has to be placed 10 to 20 m away from the phasemeter. To overcome these problems the phasemeter Interface (PMI) was developed in the here presented work. The purpose of the PMI is to provide an interface with ethernet port, that allows to control and use the phasemeter. The PMI can be controlled by a set of commands of a proprietary communication protocol. Several functions, beyond EPP low level handshakes, are provided to initialize or configure the phasemeter, in order to ease software development on the computer system. The communication is done upon a common TCP/IP protocol stack with UDP packets. It is assumed that the collision domain consists of maximum two hosts to get virtually a real time capable connection. The PMI is based on a powerful ARM9 microcontroller and an off-the-shelf board with an attached EPP extension board. The design of the extension board is part of this thesis.

# Contents


**1 Project Context** **6**
- 1.1 AEI 10 m Prototype Interferometer . . . 6
- 1.2 Suspension Plattform Interferometer . . . 7
- 1.3 Phasemeter . . . 7

**2 Hardware** **8**
- 2.1 Microcontroller . . . 8
- 2.2 Microcontroller Board . . . 9
- 2.3 EPP Extension Board . . . 10

**3 Software** **14**
- 3.1 Architecture . . . 14
  - 3.1.1 Timing and Latency . . . 15
  - 3.1.2 Ethernet Driver and TCP/IP-Stack . . . 19
    - 3.1.2.1 Receiving . . . 20
    - 3.1.2.2 Transmission . . . 21
    - 3.1.2.3 Checksum . . . 22
    - 3.1.2.4 Header Access and Assembling . . . 22
    - 3.1.2.5 Packet Descriptor . . . 23
    - 3.1.2.6 EMAC Initialization . . . 23
      - 3.1.2.6.1 Ethernet MAC . . . 24
      - 3.1.2.6.2 Buffers . . . 24
      - 3.1.2.6.3 Buffer Descriptors . . . 24
    - 3.1.2.7 PHY Driver . . . 25
  - 3.1.3 PM3 Driver . . . 25
    - 3.1.3.1 Set Table . . . 26
    - 3.1.3.2 Read Table . . . 27
    - 3.1.3.3 Recording . . . 27
  - 3.1.4 EPP Driver . . . 29
  - 3.1.5 RS232 Driver . . . 30
  - 3.1.6 Memory Management Unit (MMU) and Cache Configuration . . . 30
  - 3.1.7 Reset . . . 31
  - 3.1.8 Command Queue . . . 31
  - 3.1.9 Execute Command . . . 32
  - 3.1.10 Message Handler . . . 32
  - 3.1.11 Time . . . 32
  - 3.1.12 Interrupt Handler . . . 34
  - 3.1.13 Exception Handler . . . 34
  - 3.1.14 Assembly Routines . . . 34
- 3.2 Boot Sequence . . . 34
  - 3.2.1 Embedded Boot Program . . . 35

3.2.2 AT91Bootstrap Framework . . . . . 36
3.3 Other Initialization . . . . . 37
3.3.1 C-Runtime . . . . . 37
3.3.1.1 Stack Pointer Initialization . . . . . 37
3.3.1.2 Exception Vector Table . . . . . 38
3.3.1.3 Zero Uninitialized Variables . . . . . 38
3.3.1.4 Interrupts . . . . . 38
3.3.1.5 Start of the Main Routine . . . . . 39
3.3.2 Ethernet PHY . . . . . 39
3.3.2.1 PHY (interface) . . . . . 39
3.3.2.2 PHY (itself) . . . . . 39
3.3.3 EPP Extension Board . . . . . 40
3.3.4 Advanced Interrupt Controller (AIC) . . . . . 40
3.3.5 Reset Controller (RSTC) . . . . . 40
3.3.6 Peripheral Clocks . . . . . 41
3.4 Configuration . . . . . 41

**4 Communication** **42**
4.1 Protocol Stack . . . . . 42
4.1.1 Physical Layer (PHY) . . . . . 43
4.1.2 Data Link Layer (EMAC) . . . . . 43
4.1.3 Network and Transport Layer . . . . . 43
4.2 Size Boundaries and Fragmentation . . . . . 43
4.3 UDP Packet-Loss and Detection . . . . . 44
4.4 Status and Error Messaging . . . . . 44

**5 Using the Phasemeter through the PMI** **45**
5.1 Phasemeter Initialization . . . . . 46
5.2 Byte Order . . . . . 46
5.3 Command Acknowledgement . . . . . 46
5.4 Recording Mode . . . . . 47
5.5 Phasemeter Modification for Latency Optimization . . 47
5.6 Phasemeter Documentation . . . . . 48

**6 Compiling & Programming** **49**
6.1 Development Environment . . . . . 49
6.2 Compiling . . . . . 49
6.3 Flash Programming . . . . . 49

**A Appendix** Schematic of the EPP Extension Board **50**

**B Appendix** Memory Mapping **51**

**C Appendix** PMI Communication Protocol **52**

**D Appendix** PMI Configuration **54**

**E Appendix** Version History **55**

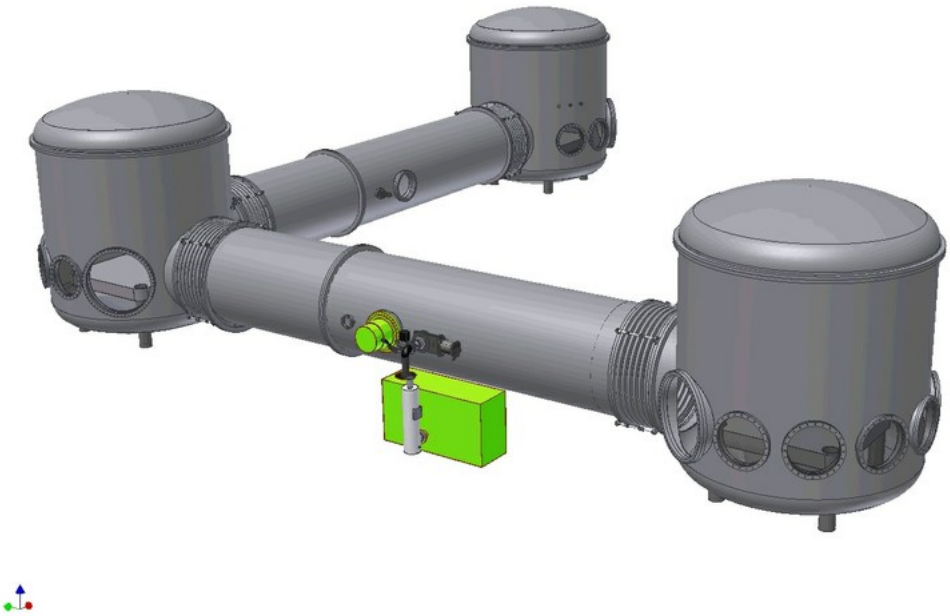


Figure 1: AEI 10 m Prototype Interferometer schematic

# 1 Project Context

## 1.1 AEI 10 m Prototype Interferometer

The 10 m Prototype Interferometer will be used to test and develop technologies for potential future upgrades of the gravitational-wave detector GEO600 [4]. Additionally, several experiments to explore quantum mechanical effects in macroscopic objects will be performed in the prototype facility. The Prototype is still under construction at the Albert Einstein Institut (AEI) in Hannover.

A schematic of the vacuum enelope for the interferometer can be seen in Figure 1. It consists of three tanks, which are connected by two beam tubes. The L-shaped system has an arm length of around 10 m and encloses about 100 $m^3$. Each tank has a diameter of 3 m and a height of 3.4 m, the diameter of the tubes is 1.5 m. The system is made of about 22 t of stainless steel. Each of the three tanks contains a suspended, seismically isolated table. These tables carry all vacuum-sided mechanics and optics, e.g. the Suspension Platform Interferometer.

## 1.2 Suspension Plattform Interferometer

The Suspension Platform Interferometer (SPI) consists of a set of auxiliary interferometers to measure the relative motion between the seismically isolated optical tables of the 10 m Prototype. The purpose of the SPI is to control the differential longitudinal displacement of the tables with an aimed accuracy of only $100\,\mathrm{pm}/\sqrt{\mathrm{Hz}}$ at 10 mHz. The measured variables, the longitudinal displacement for instance, are used to derive control signals that are applied to the actuators of the tables.

All three translational degrees of freedom as well as pitch and yaw, are measured by the SPI interferometrically. The SPI consists of three heterodyne Mach-Zehnder interferometers. One of them is used to create a reference point for the phase measurement and the two others are to measure the motion between the tables relative to the central table. The south and west table are situated 11.65 m from the central table (center to center), respectively. To achieve a control bandwidth of 100 Hz, a heterodyne frequency of about 20 kHz has been chosen. This set of interferometers is read out by means of a LISA pathfinder type phasemeter.

## 1.3 Phasemeter

The phasemeter is an essential part of the SPI. It performs the following steps: converting photo current coming from the SPI photodiodes into voltage, digitizing those by a 16 bit A/D converter at a sampling frequency of 800 kHz, and performing a single-bin discrete Fourier transform (SBDFT) [38] to gain phase information of the signal which is a measure for the relative distance of the tables. All these steps are performed independently for each of the 20 phasemeter channels, related to signals from up to five quadrant photodiodes.

# 2 Hardware

The choice for an appropriate microcontroller and -board was determined by the following facts:

- **High performance** to achieve low latency and to allow subsequent extensions.
- **Embedded peripherals** to avoid external components, to ease software development and to achieve a better performance.
- **Extension capability** to attach an EPP extension board.
- **Supported by open source tools** for development and debugging.

To limit complexity and costs, the decision was taken to use a ready made development board instead of an own design. Because that all available boards lack an EPP interface, the development of an extension board became necessary.

## 2.1 Microcontroller

The decision was made in favour of an *ARM9*-based microcontroller in general and an Atmel *AT91SAM9260* in particular. The *ARM9* core provides enough performance and the powerful peripherals of the Atmel microcontroller unit (MCU) meets the demands well. Atmel is besides *NXP-Semiconducter* one of the most popular manufacturer of *ARM9* MCUs.

The key benefits of the *AT91SAM9260* type are listed below:

- ARM926EJ-S core
- Harvard architecture
- Memory Protection Unit (MMU)
- 2 x 8 kB data and instruction cache
- Up to 200 MHz operation
- IEEE 1149.1 JTAG Boundary Scan on all digital pins
- 2 x 4 kB SRAM
- Ethernet MAC 10/100 Base T

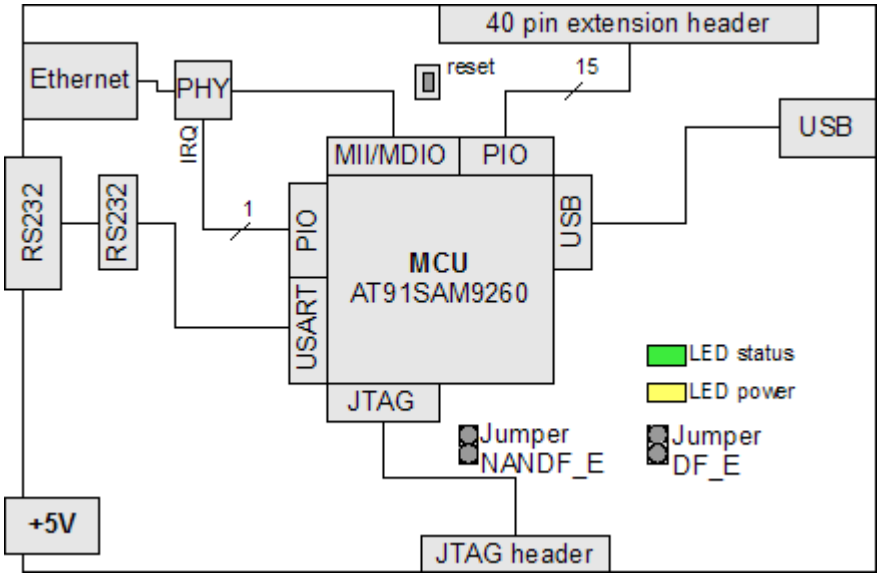



Figure 2: Schematic diagram of the MCU board with used peripherals

- Peripheral DMA channels
- Four Universal Synchronous/Asynchronous Receiver Transmitters (USART)
- Three 32 bit Parallel Input/Output Controllers (PIOC)
- A wide range of off-the-shelf boards available
- Supported by open source tools at all points: GCC, OpenOCD, etc.

## 2.2 Microcontroller Board

The chosen board is an *Olimex SAM9-L9260* type. It is very similar to the Atmel *AT91SAM9269-EK* evaluation kit, so software for this board can also be used for the Olimex one. Additionally, there are several projects done upon this board and documented in the internet.

The board has one essential feature: it provides an extension port with all unused I/O-pins and thereby enough I/O lines to attach the EPP extension board.

The key benefits of the board are listed below:

- 180 MHz CPU clock with 90 MHz system clock
- 64 MB SDRAM

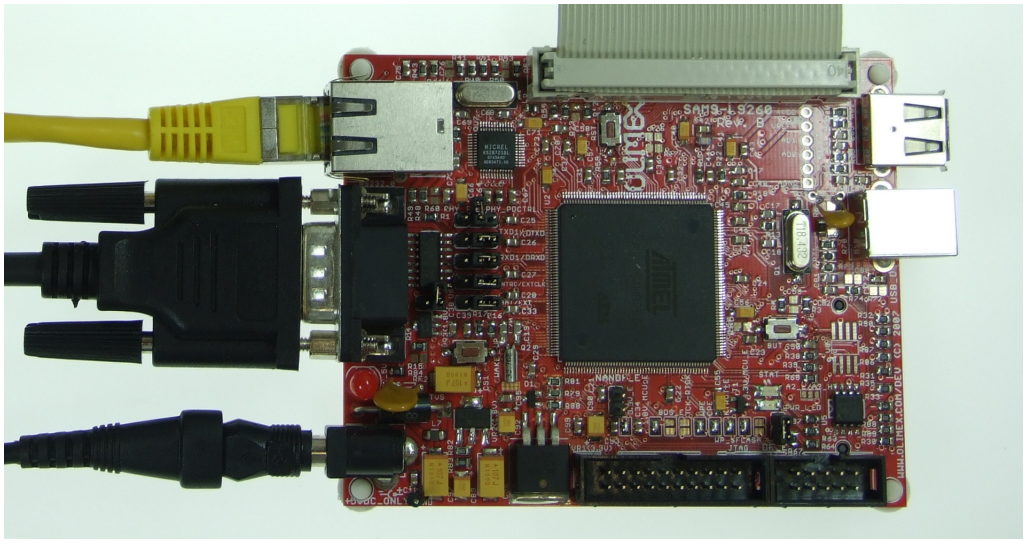


Figure 3: Microcontroller board

- 2 MB DataFlash
- RS232 interface
- Ethernet interface
- USB interface (for in-system-programming)
- JTAG interface (debugging)
- Single power supply of 5V
- 40pin extension header

## 2.3 EPP Extension Board

The purpose of the EPP extension board is to add an *IEEE1284v2* conform Enhanced Parallel Port to the MCU-board. In a logical respect, the peripheral device, in this case the phasemeter, could be connected directly to the MCU. But electrical adjustments are necessary, what makes the extension board essential. For example, the cable signal voltage, which has to be tolerated by the bus drivers with up to 7 V, must be reduced to 3,3 V for the MCU. In addition, to have some IC's between the cable and the MCU, protects from short-circuit and overvoltage.

The EPP handshake is performed by software, using the MCU's Peripheral I/O Controller[17] (PIOC). The 8 bit data and address bus, as well as

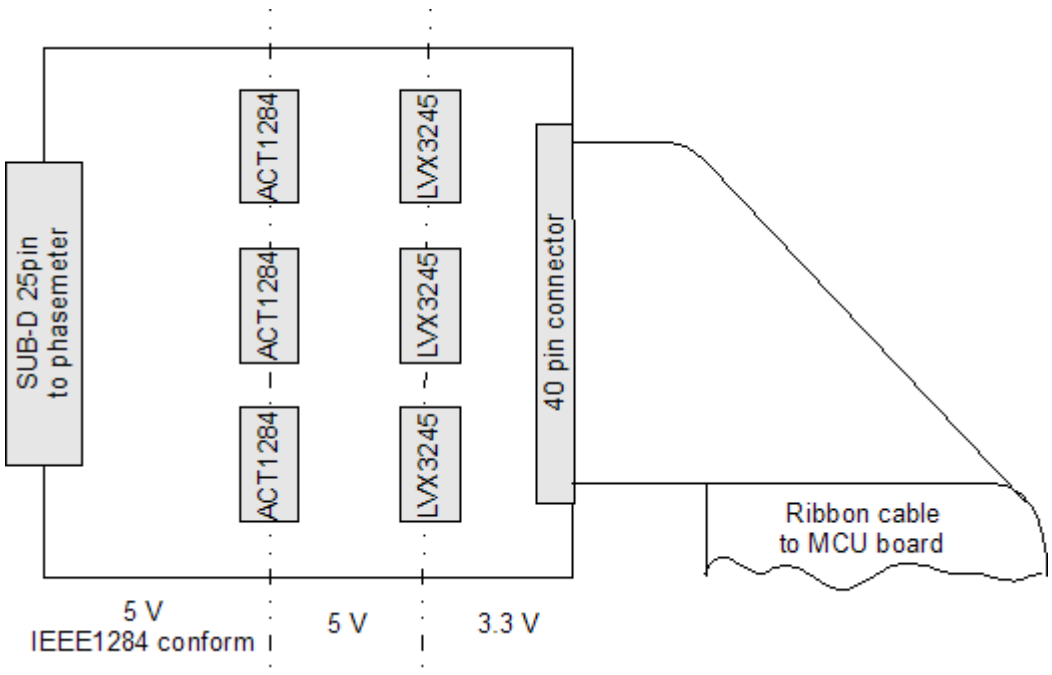



Figure 4: Schematic diagram of the EPP extension board

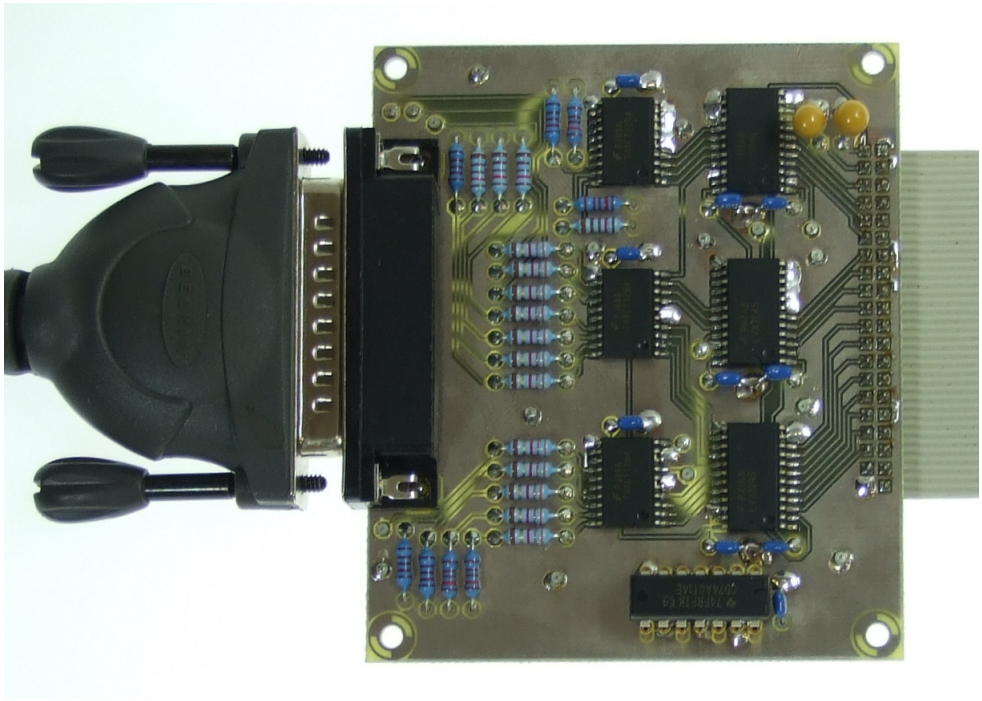


Figure 5: EPP extension board

additional control and status lines, can be accessed by a write or read operation to the corresponding 32 bit PIOC register. The data-lines are connected to the PIOC pins in a manner, that no alteration of order is necessary.

The simplest way to convert the signals between the cable and the MCU is to use one *74LVX1284* IC, which is made by several manufacturer like Texas Instruments, for example. These chips are only available in a *TSSOP* box or similar, which cannot be soldered by the electronic workshop of the Albert-Einstein-Institute (AEI). To avoid an expensive external production, the decision was taken to design a board which can be entirely made by the AEI.

The current EPP-board design is based upon three *74ACT1284* IC's which are also dedicated for *IEEE1284* issues. The disadvantage of those is that three instead of one IC is needed to cover all EPP-lines. To convert the 5 V signals of the *74ACT1284*'s to 3.3 V for the MCU, three more translation bus driver of a *74LVX3245*-type are necessary. In addition, one more IC, an inverting bus driver of the *74ACT14*-type and a several resistors to adjust the output impedance to 50 Ω are required, which are already inbuilt in the *74LVX1284*.

As already mentioned above, the design makes use of chips of two logic subfamilies of the 7400 series. In such a mixed design, one have to assure whether the different subfamilies are compatible among each other. The internet page *www.interfacebus.com*[22] covers this subject.

The circuit of the EPP extension board can be found in Appendix A. The design solution is inspired by *The Parallel Port Complete*[18]. Some lines of the *74ACT1284*'s and all of the *74LVX3245*'s are bi-directional. The working direction can be controlled by a corresponding pin[25, 28]. The current data-flow direction is determined by the EPP *write*-line. The signal has to be inverted for the direction pins of the bus-driver by one line of the *74ACT14* IC. Additionally, this signal needs to be converted into a 3.3 V level for and by one of the *74LVX3245* IC's.

It is very important, that the PMI asserts the EPP *write*-signal, before the PIOC-lines of the data bus are switched to output mode. The reason is that both, MCU and bus-drivers run in three-state mode (push-pull with high impedance state). When two such drivers in output state are connected to each other, high current due to a short circuit, fatal reflections and also damage can be the result.

The *data strobe* and *wait*-line control the data transfer. To avoid that disturbances like reflections are considered as a logical puls by the PMI, the PIOC-glitch filter is enabled in the MCU for the *wait*-line. A glitch can

potentially lead to miscount, if the *wait*-line ist affected.[1] (This issue is also discussed in Section 3.1.3.3). A disturbance affecting the *data strobe* can also lead to miscount, if the phasemeter considers the glitch as a logical signal. Unfortunately, the phasemeter has no glitch filter and the described problem in fact occurs from time to time. Another sollution is, besides the glitch filter, to reduce the slope rate of the signals, to avoid reflections and to filter high frequencies. A successful approach is to attach capacitors to all EPP-lines on the EPP-board against ground, as it is done on the board which can be seen in Figure 5. The capacitors have a value of 470 pF. They are not mentioned in the circuit in Appendix B, because that the use of capacitors is meant as a proposal.

[1] Miscount means, that the counter in the PMI which counts the read bytes of a data block, is not coherent to the number of bytes which are already provided or sent by the phasemeter. This problem occurs, when a glitch simulates that the PMI want to read a byte or when the PMI thereby thinks that the requested byte can be read.

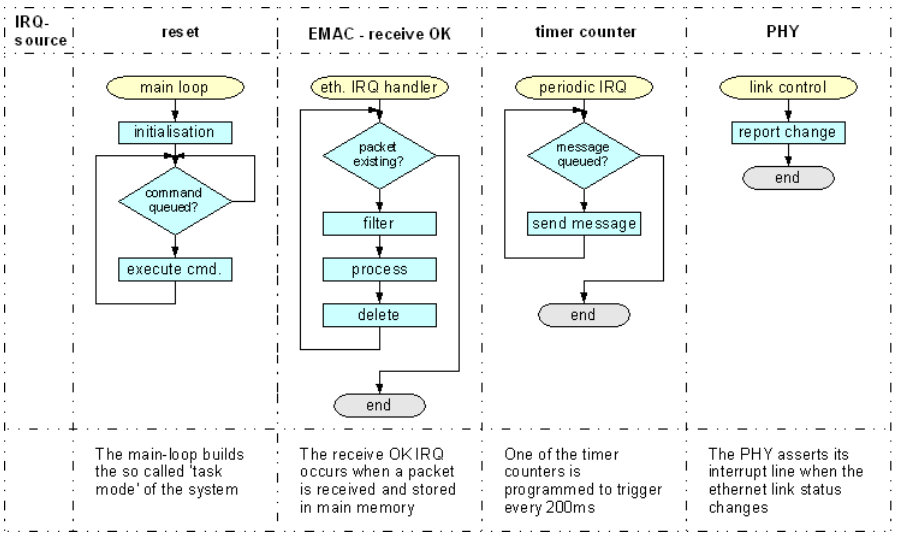



Figure 6: Task mode and interrupts of the PMI

# 3 Software

## 3.1 Architecture

The PMI is designed as a so called *bare metal system,* what means, that no operating system or at least a scheduler is used. As can be seen from Figure 6, the system consists on the one hand of a main loop (infinity loop), which builds the so called *task mode* of the system, and on the other hand, of several interrupts with associated interrupt handlers. The functionality of the PMI is command-based, what means that the PMI waits for commands via ethernet and processes them consecutively. The command processing is mainly done in task mode. There, a command queue is polled and, when a command is gotten, the *execute()* routine is run to process the command. Not before a command is finished, the next will be treated. In such a *round robin with interrupts*-architecture[52], it is not possible to handle different threads concurrently in task mode like in a scheduling-architecture. However, this technique meets the requirements regarding speed and latency best. A scheduling functionality would either beeing unused or would increase the system latency by providing time slices to other tasks. For more information about system architectures for embedded systems, refer to *An Embedded Software Primer*[52].

Figure 7 shows all software modules at a glance. By means of the colors yellow and green, the processor mode, either task or interrupt mode, can be distinguished. Some software modules are of a reentrant or a so called

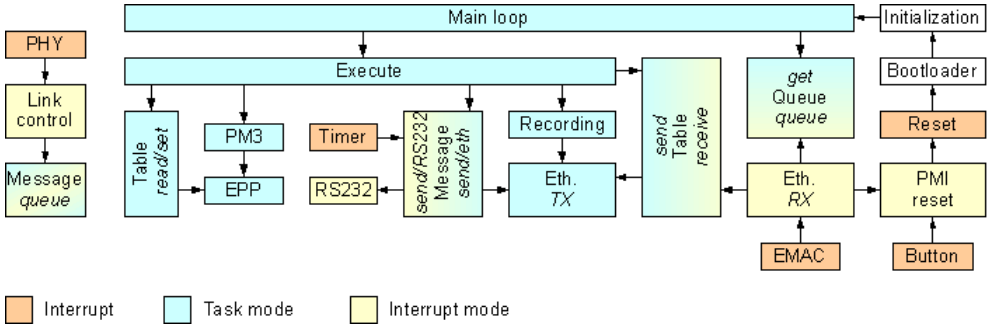



Figure 7: Overview about the software modules and their relation to each other

*pure* design, so that invoking such a routine from interrupt mode, while it is still executed in task mode, do no harm. For example, the *message()* routine is used by several routines running in both modi. For further reading about reentrancy issues, refer to the introduction by *Jack Ganssle*[36] or *An Embedded Software Primer*[52]. The orange colored boxes are interrupt sources and the white ones are not part of the PMI software in a narrower sense, but rather a part of the bootloader.

### 3.1.1 Timing and Latency

The recording mode is the normal operation mode of the PMI, besides the phasemeter initialization mode. In the recording mode, the PMI reads measurement data from the phasemeter block-wise and sends a read block immediately to the computer system via ethernet. This mode is time critical and has hard realtime requirements. The time to send a data block must be short to keep the latency low. The time which is needed to read one data block from the phasemeter and the time to send it, defines the maximum recording speed.

As can be seen from Figure 8, the PMI is part of a control loop of the Suspension Plattform Interferometer (SPI). Both, latency and transfer rate are critical factors regarding control bandwidth of the loop. Since the latency of the other components is not fully determined yet, the latency and transfer rate requirements of the PMI are not exactly defineable. But the lower the latency of the PMI, the higher the latency margins of the other components of the loop. Also very important is that the latency should be stable, which is hard to guarantee, since the PMI has to handle potentially occuring network packets in the recording mode (which are not part of the PMI communication). But since the PMI and the computer system are the only hosts in the ethernet collision domain, it is the task of the computer

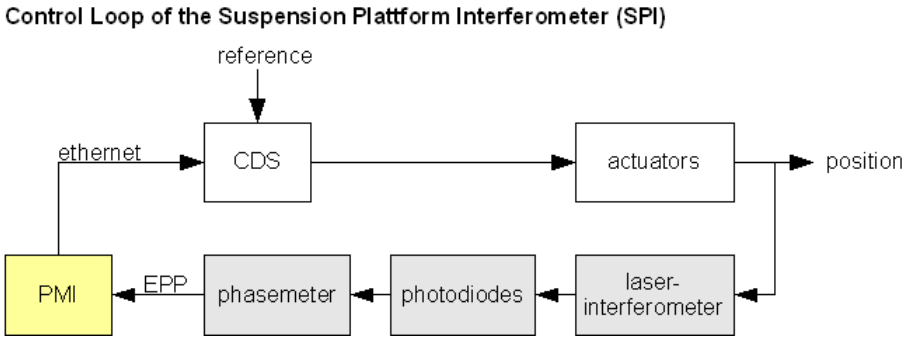



Figure 8: Schematic of the control loop of the Suspension Platform Interferometer (SPI)

system (-programmer) to avoid any not relevant packet transmission. If this can be assured, the PMI will never receive a packet during recording mode except, when it has to stop the acquisition anyway by receiving the *Reset* or *Stop* command.

As already mentioned, only the recording mode is time critical and only the involved software modules, namely the ethernet driver and TCP/IP-Stack as well as the recording module, need to be optimized regarding speed and latency. For such optimization, it is worth to be aware of the phasemeter timings, to know the limits there.

The phase-measurement-data is generated by up to 20 extension cards of the phasemeter, which perform a Discrete Fourier Transformation (DFT) to reduce the amount of data. The result is a data block which carries the measurement data besides additional information and which has a length of 22 B[37, 39]. The data blocks are read cyclically, card by card, through two serial buses (each bus for a batch of 10 cards) by the main-board of the phasemeter. The serial buses are clocked with 20 MHz, which equates a tranfer rate of 2.5 MB/s per bus and a data-block-rate of around 11 kHz at 20 channels. This is obviously not a point to worry about. The next station to be considered is the FPGA on the main-board, which forwards the data to the FIFO. The data is transmitted bytewise with a clock of 800 kHz and thus 800 kB/s. This is in fact a weak point. As can be seen from Figure 9, the phase-data-rate is thereby limited to 1.8 kHz at 20 channels.

Since the data-rate of the FIFO's input is known to be 800 kB/s, the period of providing one data block of 20 channels can be calculated and is 556 µs. The PMI must be able to read and send the data blocks with at least the same frequency as the phasemeter provides it, to avoid a buffer overflow in the FIFO. Even a short transgression which would be compensated by an

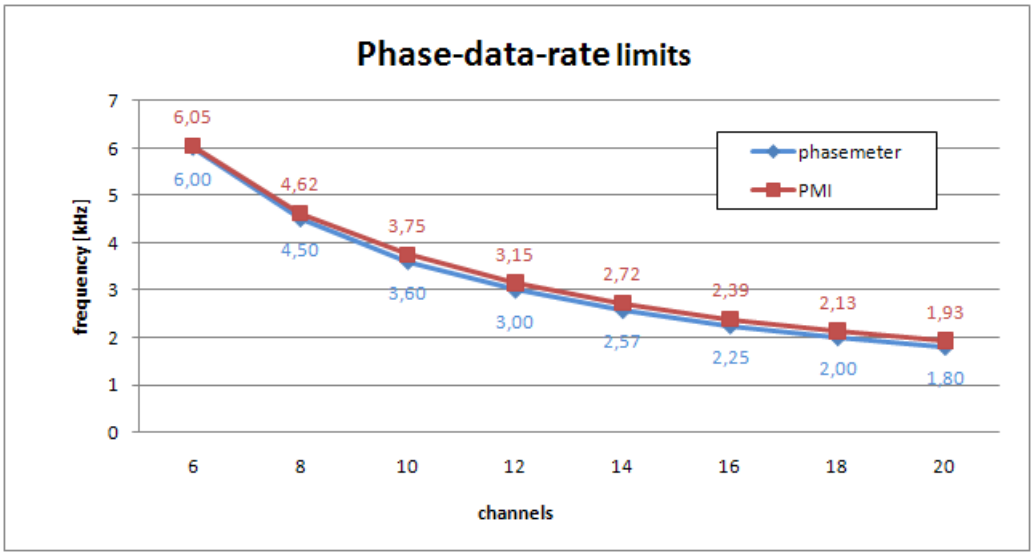



Figure 9: Theoretical maximum phase-data-rates of the phasemeter and PMI.

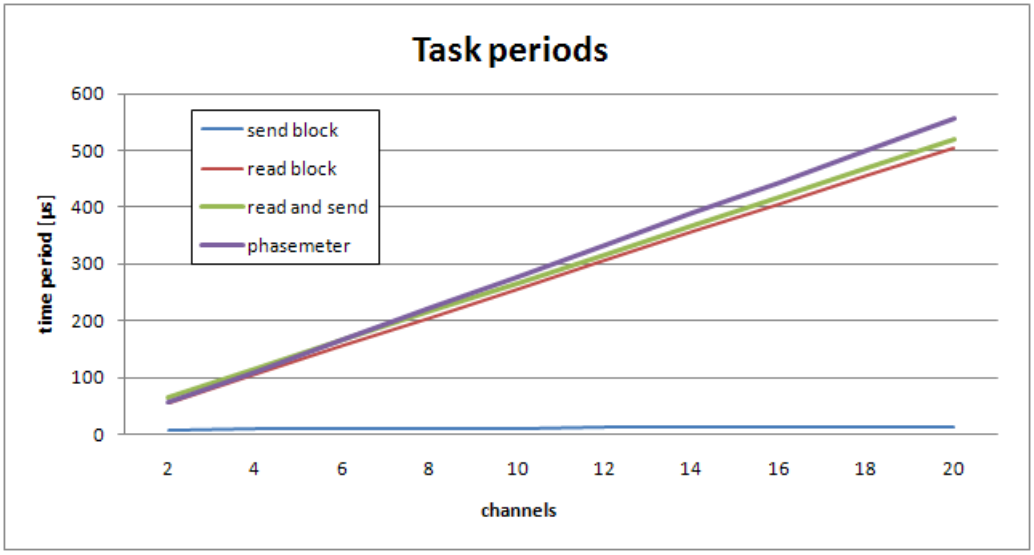



Figure 10: Periods of the PMI for reading, sending and both as well as the period of loading the phasemeter's FIFO with a phase-data-block.

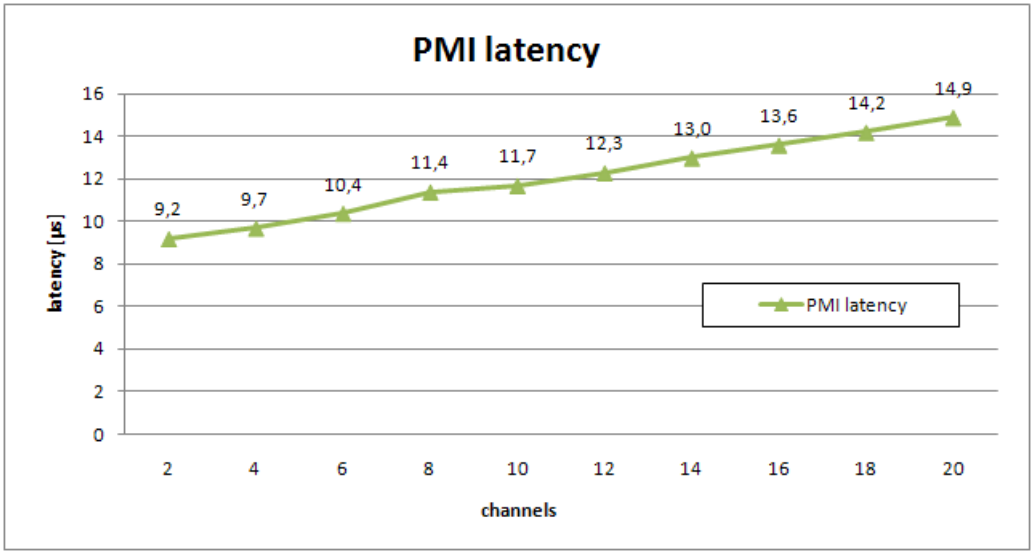



Figure 11: PMI latency

average data-rate higher than the phasemeter's must be avoided, because it will increase and destabilize the latency. The period for the PMI to read and send a data-block is composed of the period to read the data-block from the phasemeter and the period to send the block via ethernet. The transmit and read timings, dependend on the number of channels, are shown in Figure 10. Putting everything together, the result gives the maximum phase-data-rate for phasemeter with attached PMI, as shown in Figure 9. It is important to note, that the measurement is done with connecting the PMI directly to the phasemeter (without a cable inbetween). Since the signal slope rises due to cable capacity, the transmission speed may decrease slightly.

The latency of the PMI in microseconds can be seen in Figure 11. The ascertained latency is the time between the last byte of a data block, written into the FIFO of the phasemeter, and the beginning of the ethernet data transfer (the time where the first byte appears on the cable). Three time periods are taken into account: First, the time between the write of the last byte into the FIFO and the completion of the EPP handshake, which reads this byte. Second, the time between the completion of the handshake and the MII data transfer of the packet to the PHY (Section 3.3.2). And third, the latency of the PHY which is mentioned in the data sheet[46].

The first time period is of around one microsecond, irrespective how many channels are covered. The reason for that is, that the PMI reads the bytes faster than they are written into the FIFO, thus at the end of every data block read cycle, the PMI always has to wait for the next byte and reads it

immediately out of the FIFO. The third period, the PHY latency, is marginal and according to the PHY data sheet of around 90 ns. The decisive factor is the second considered period, which is bred by the ethernet driver and TCP/IP stack.

### 3.1.2 Ethernet Driver and TCP/IP-Stack

*Source files: eth_ init.c, eth_ init.h, eth_ tx.c, eth_ tx.h, eth_ rx_ irq.c, eth_ rx_ irq.h, eth_ rx_ packet.c, eth_ rx_ packet.h, eth_ chksum.c, eth_ chksum.h, eth_ phy.c, eth_ phy.h*
The ethernet driver and TCP/IP-stack together build the most complex and extensive part of the PMI. To reach a maximum of processing speed, both are tightly coupled and can be regarded as a single software module. The module, which is in the following mostly called *ethernet driver*, provides the following features:

- PHY driver
- MCU EMAC driver
- Custom-made ligth-weight TCP/IP stack
- IP and UDP checksum calculation
- Ethernet II, IPv4, UDP support
- Packet filtering
- PMI communication protocol support
- High speed and low latency operation

As already mentioned, the ethernet driver is also used by the recording module and is thus time critical. The phase-data-block from the phasemeter must be sent as soon as possible, to keep the latency low and to leave as much time as possible for the recording module which reads the data-block. The processing time of both defines the transfer-rate limit, as can be seen in Figure 10. The high requirements regarding speed and latency lead to the development of an own TCP/IP-stack and EMAC driver instead of using a ready made one like *LwIP*[5] as TCP/IP-stack and the EMAC driver which is provided by *Atmel*[1], for example. The PMI ethernet driver (including the TCP/IP-stack) provides functions which make time consuming buffer copying needless by storing to be sent data directly in the transmit buffers of the MCU and in particular the EMAC.

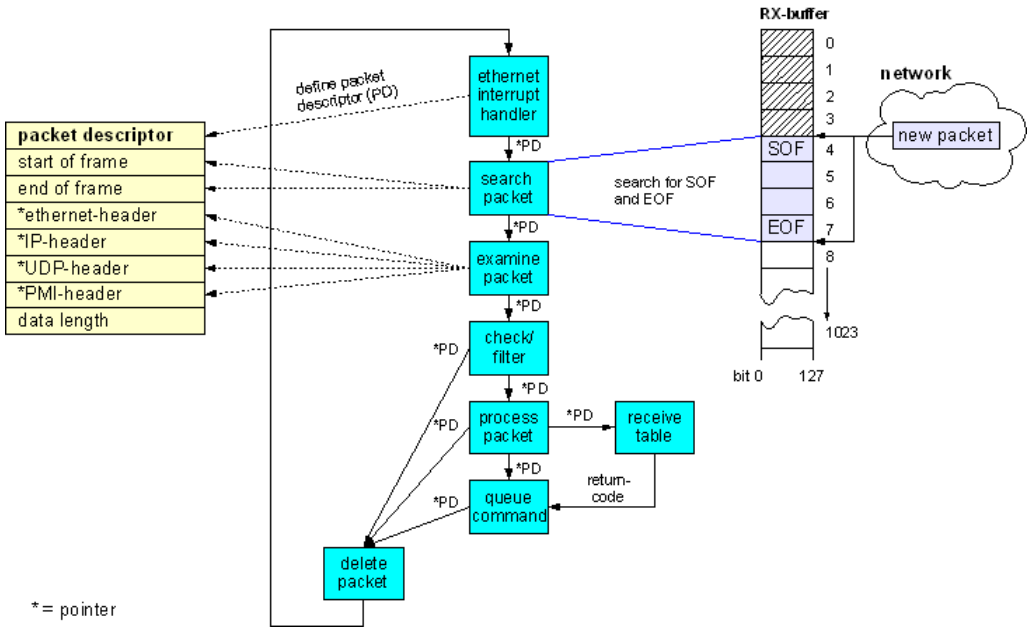



Figure 12: Receiving functional diagram

#### 3.1.2.1 Receiving

*Source files: eth_rx_irq.c, eth_rx_irq.h, eth_rx_packet.c, eth_rx_packet.h*

The receiving sequence is outlined in Figure 12. The complex receiving scheme of the EMAC and its buffer management is out of the scope of this manual. For information about this issue rather refer to the MCU manual[17]. The only thing which is important to know to understand the software architecture is, that a received packet is stored in one or more 128 B buffers, depending on the length of the packet. In the example in Figure 12, the received packet is stored in four buffers. The PMI is configured to use the maximum of 1024 receive buffers. The buffers are used cyclically, which means that a buffer pointer in the EMAC is incremented on every used buffer and reset to zero when the last one is reached.

The *eth_rx_search_frame()* routine starts at buffer number zero, like the EMAC, and remembers the position of the last packet. It is invoked by an *EMAC receive OK* interrupt and looks for the *start-of-frame* (SOF) and *end-of-frame* (EOF) flags, which enclose a packet. If a packet is found, the numbers of the first and the last buffer of the array is registered in the packet descriptor *RxPd*. A pointer to the *RxPd* is again used as a parameter for all the following routines, which handle and process the packet.

The next step is to apply a header-mask to the first buffer, which contains the protocol headers, by the *eth_rx_header()* routine. The task of the mask

is, that the different header fields of the different data types can easily be accessed through a C-structure, which is much more convenient than dealing with pointer-offsets and type-castings. The packet descriptor is described in Section 3.1.2.5 and is the same for receiving as well as transmit. The fields of the descriptor are shown in Figure 12. The mask consists of four structs which describe the ethernet, IP, UDP and PMI headers. The *eth_rx_header()* routine simply sets the struct-pointers in the packet descriptor to the appropriate location of the receive buffer. For example, the IP header is located 14 B ahead from the start of the buffer, because of the length of the ethernet header of 14 B. From now on, all header fields are accessable by the struct members of the four structs.

In the fourth step, the filter routine *eth_rx_check()* will discard those packets, which are not part of the PMI communication or simply corrupt. For that, on the one hand, several header fields are validated to distinguish whether the protocol structure is expected, and on the other hand, the IP and UDP checksums are validated to detect damaged packets.

All valid packets will then examined by the *eth_rx_process()* routine. The routine first looks for commands which should be processed by the interrupt handler itself, rather than the main loop. These packets contain either priority commands or the *Set Table* command, in which case a table fragment is stored in the packet which again has to be copied to main memory by the *pm3_table()* routine. The priority commands are *Reset* and *Stop*.

Those commands, which are not processed here or do need additional handling (the *Set Table* command, after receiving the whole sin/cos table), are queued in the command queue, which is polled by the main loop. After all, packets which are processed, queued or dropped, will be deleted to deallocate the buffers by the *eth_rx_delete()* routine.

#### 3.1.2.2 Transmission

*Source files: eth_tx.c, eth_tx.h*

The transmit sequence can be seen in Figure 13. If any software module wants to send data via ethernet, it first has to define a packet descriptor called *TxPd*. The next step is to allocate a free transmit buffer by calling the *eth_tx_allocate()* routine of the ethernet driver. This routine expects a pointer to *TxPd* and will there register the number of the buffer which should be used for the data. The data will be copied directly into the transmit buffer to avoid additional copying. After the (optional) data is copied and the fields in the PMI header are set, the *data_length* field of the *TxPd* has to be set before the *eth_tx_send()* routine can be invoked, also with a pointer to *TxPd*. This routine copies the ethernet, IP and UDP headers into the

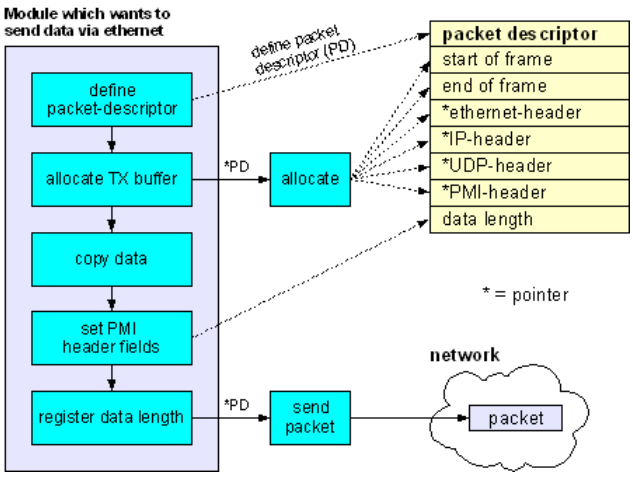



Figure 13: Transmit functional diagram

transmit buffer, computes the IP and UDP checksums (the ethernet one is calculated and appended by hardware and namely by the EMAC) and sends the packet, after all.

Allocating a buffer involves that the EMAC will wait for the first allocated buffer until a specific flag is set, which indicates that the buffer should be sent. If a routine allocates a buffer but will not use it anymore, the routine has to de-allocate the buffer by calling the *eth_ tx_ deallocate()* routine. The EMAC will never automatically skip an unused buffer and will thus wait forever.

#### 3.1.2.3 Checksum

*Source files: eth_ chksum.c, eth_ chksum.h*

All used network protocols, namely Ethernet II, IP and UDP, use checksums. The Ethernet II checksum is calculated by the EMAC. The IP and UDP checksums are computed by corresponding subroutines. These subroutines rely on another subroutine, which implements the actual checksum algorithm. Mainly the UDP checksum computation is very time consuming, because that in contrast to IP, the payload is also covered. Several techniques are used to speed up the calculation, which are described in *Computing the Internet Checksum*[49]. A good introduction about the one's complement algorithm gives the article *Compute 16-bit Ones's Complement Sum*[3].

#### 3.1.2.4 Header Access and Assembling

*Source files: eth_ descrpt.c, eth_ descrpt.h*

A network protocol header consists of several different fields with different data types[53]. To access these fields straightforward, a struct can be used,

which includes all header fields in the same order. But to increase the access speed, compilers do allign variables on a boundary which equals the size. That means that a 32 bit integer would be leaded by a two byte padding, if a 16 bit-integer precedes it. That can be avoided by using an attribute-directive of GCC:

```
__attribute__((packed))
```

Such a packed struct does not contain any padding and can thus be copied in one piece to a transmit buffer or can be used as a mask to access header fields of a received packet.

The receive as well as the transmit buffers are arrays of unsigned chars. By superposing a header struct with such an array, a pointer aliasing issue arises. ISO-C specifies, that pointer of a different data type must not point to the same location. This constraint is used by compilers for optimization, what then leads to undefined values after dereferencing a casted pointer. GCC uses aliasing rules in optimization levels -O2, -O3 and -Os. Because there is no other rational way to access or assemble headers, the maximum optimization level for associated files is -O1. It is recommended to use -O1 for all files. Refer to *Krister Walfridsson*[56] or *Using the GNU Compiler Collection*[31] for further information about pointer aliasing.

#### 3.1.2.5 Packet Descriptor

*Source files: eth_descrpt.c, eth_descrpt.h*

The packet descriptor is an uniform data container which contains all relevant information about a packet which can be existent in memory as a single data stream in the case of received packets or splitted in the case of to be transmitted packets. All the modules of the ethernet driver expect a pointer to a packet descriptor as parameter, except those which expect no parameter at all. The construction can be seen from Figure 12 and 13. It consists of two integer, which store the first and the last buffer number of the packet[2], several struct pointer, which point to the different headers, and another integer to store the data length, which is appended to the PMI protocol header.

#### 3.1.2.6 EMAC Initialization

*Source files: eth_init.c, eth_init.h*

The EMAC as well as associated buffers and descriptors are set up by this module. It can be divided into the following two parts:

- Ethernet-MAC (EMAC) configuration
- Buffers and descriptors

[2] Compared to the transmit buffers, one receive buffer of only 128 B cannot store a larger packet which therefore must be splitted and stored in multiple buffers. A transmit buffer can hold a packet with its maximum length of 1514 B.

**3.1.2.6.1 Ethernet MAC** The Ethernet MAC (EMAC, described in Section 4.1.2) needs to be configured before any communication can take place. Information about the link speed, duplex mode and protocol configuration must be set. The EMAC needs also to know, where the buffer descriptors lay in memory. In addition, some control flags must be set to get the EMAC working. To set up the EMAC, you have to write to the corresponding registers which are mapped into the memory space. Refer to the MCU data sheet [17] for further information.

**3.1.2.6.2 Buffers** For received as well as to be transmitted packets, buffer arrays in main memory are needed. The transmit buffers have a variable length of up to 2047 B, receive buffers of only 128 B. A maximum number of 1024 buffers for each direction has to be defined in fast memory. The board is equipped with 64 MB SDRAM and has additionally two 4 kB SRAM memory blocks inbuild. The SRAM size is too small to define a sufficient number of buffers there, so we have to use SDRAM. A data access on SDRAM takes a few clock cycles more time in comparison with SRAM. However, this should be no problem, because SDRAM is nevertheless fast enough for this purpose. But in the errata list of the MCU manual[17] is mentioned, that in some circumstances buffer underruns may occur if SDRAM is used. This has never been noticed in this project.

The PMI uses 1024 transmit buffers, each with a length of 2047 B and 1024 receive buffers, each with a length of 128 B.

**3.1.2.6.3 Buffer Descriptors** Each buffer has its appendant a buffer descriptor. The descriptors have a length of two words (eight bytes) and can be divided into two parts. The first word contains the start address of the coresponding buffer and the second control and status information. Only in the receive buffer descriptors are the two least significant bits of the address field used as flags, too. Because of the four byte memory allignment[3], those two bytes of an address were anyway always zero. The buffer descriptors of each direction have to lie consecutively in memory, because the EMAC knows only the address of the first descriptor and increments this pointer to come to the next descriptor. The address must be set for each direction in the *Buffer Queue Pointer Register* of the EMAC. An internal counter counts up to 1023 and resets afterwards itselves to zero. For this reason, a cyclic buffer management is required, to use the buffers circularly.

[3]The MCU can only access data, which is aligned on a boundary which equals the data unit size. Thus, a four byte integer needs a four byte alignment and a two byte integer, a two byte alignment, for example.

Before the buffers can be used, the buffer descriptors must be initialized. For the transmit buffer descriptors the *used*-bit must be set and for the receive buffer descriptors the *ownership*-bit must be reset. The *ownership*-bit flags that the corresponding buffer is used to store a received frame and needs to be zero for the EMAC to write data to the corresponding buffer. The *used*-bit is set, when the buffer contains a frame which have to be transmitted by the EMAC. In case that less than 1024 buffers are defined, the *wrap*-bit must be set in the descriptor of the last buffer.

#### 3.1.2.7 PHY Driver

*Source files: eth_phy.c, eth_phy.h*

The PHY[4] driver provides functions to configure and control the PHY. During the PMI initialization phase, a routine is called, to set up the network parameter and to configure and enable the link up/down interrupt in the PHY. The PHY is thereby configured to assert its IRQ line, which is connected to a pin of the Parallel I/O Controller[5] (PIOC) of the MCU. The assertation of the interrupt line causes the generation of an interrupt in the MCU, by which again the PIOC interrupt handler is invoked. The interrupt is used to signal, when the ethernet link goes up or down. A link status change is announced by a message (Section 3.1.10).

### 3.1.3 PM3 Driver

*Source files: pm3.c, pm3.h, pm3_table.c, pm3_table.h, pm3_rec.c, pm3_rec.h*

The PM3 driver provides phasemeter specific functions, to initialize and run the phasemeter. It relies on the EPP driver (Section 3.1.4), since the PMI communicates with the phasemeter via EPP. If another interface should be used, the EPP driver can be exchanged by an appropriate driver. The provided functions are:

- Set RAM Address
- Reset RAM Address (set RAM address to zero)*
- Set RAM Data
- Read RAM Data
- Set Channels
- Set NFFT

[4]PHY means the physical layer, which is implemented in an extra hardware connected to the MCU. Refer to Section 4.1.1 for further information.

[5]The MCU includes three 32 bit general purpose parallel I/O controller.

- Set Table
- Read Table
- Set PIR
- Start Recording
- Stop Recording
- Data Unit (returns the size of a phase-data-block)*

*Not accessable through PMI commands (refer to Table 5 in Section 5).

*Read Table* and *Set Table* are more complex functions, because that the sin/cos-table has a bigger size than just a few bytes. The table is required by the phasemeter for the Discrete Fourier Transformation (DFT) and can have a size of more than 100 kB, depending on the number of supporting points[37, 39].

#### 3.1.3.1 Set Table

*Source files: pm3_table.c, pm3_table.h*

The set table module consists of two subroutines, whereby the one is to receive the sin/cos-table from the computer system and the other to send the table to the phasemeter.

When a packet with the *Set Table* command is received, the command with its data will not be queued in the command queue as it is the case for other commands. Rather, the data is immediately copied to main memory by the *pm3_table_receive()* routine[6]. The reason for this approach is, that the data field in a command queue entry is to small to hold up to nearly 1500 B of table-data of an UDP-packet. The *pm3_table_receive()* routine first checks whether the currently processed packet contains the first table-fragment. If so, the length of the table is stored and a counter is set with the same value to count the remaining bytes of the table. The counter will be compared with every received table fragment to ensure, that the fragments are assembled in the right order. In the next step, the fragment will be copied to a table buffer. To do that, the routine has to handle several 128 B receive buffers, in which the whole packet is stored. For the first buffer of a packet, the position of the fragment has to be determined first, because the data is

[6]**Timing Issues** It is unusual to run such tasks within an interrupt. But in this case, we don't have to bother whether the system latency increases, because there is no time critical task at this time. And during the (time critical) data recording mode, all none-priority commands are rejected anyway.

preceded by headers and additional command-specific information like the amount of remaining bytes. If the table is entirely received, the completion is signalled by writing the table size into a specific variable of the table descriptor. As recently as the last fragment is received and the whole table is stored, the *Set Table* command is queued, to run the *pm3_table_set()* routine in task mode (through the main loop), which sends the table to the phasemeter finally. This subroutine is quite simple, since the sin/cos-table is already stored in the table buffer. It just consists of a loop which sends the table bytewise to the phasemeter.

The table descriptor consists, besides the table buffer, of a batch of variables to signal the current state of the table receiving/setting and reading/sending-back sequences. If anything goes wrong while the computer system sends a table to the PMI and the PMI sends none or a negative acknowledgement, it is necessary to reset the table transmission procedure by sending the *Stop* command. Therewith, the table descriptor is initialized again and the transmission can be restarted. The PMI will send a message when a table is received, set, if an unexpected fragment is received or if the table transmission is reset by the *Stop* command.

According to the PMI communication protocol, the *Set Table* command will be acknowledged with a return code.

#### 3.1.3.2 Read Table

*Source files: pm3_table.c, pm3_table.h*

To verify whether the table is correctly transmitted and set into the phasemeter, the *Read Table* command of the PMI can be used to read the sin/cos-table back and to compare it with the original one.

The *Read Table* command consists of two subroutines: The *pm3_table_read()* routine reads the table from the phasemeter and stores it into the table buffer. And the *pm3_table_send()* routine sends the table piece by piece back to the computer system. The fragment size is defined to be 512 B and can be changed in the *config.h* file.

#### 3.1.3.3 Recording

*Source files: pm3_rec.c, pm3_rec.h*

The recording module consists of five subroutines, which are listed below:

- *pm3_rec()* - this function is called to start the recording mode
- *pm3_rec_start()* - initialization
- *pm3_rec_recording()* - the actual recording routine
- *pm3_rec_reset()* - to reset the data transfer after an error

- *pm3_rec_stop()* - stop recording procedure

The *pm3_rec()* routine is the only external called routine of the module, which controls the initialization, recording and stop procedure. The first thereby called subroutine is *pm3_rec_start()*, which initializes several variables and sets the phasemeter to recording mode. If this is successfully done, the *pm3_rec_recording()* routine is called to read a phase-data block and to send it via ethernet. To ensure the validity of the read data, the four startbytes, which precede each data block, are monitored. If one of those does not equal 0xff, the whole block is discarded and the return code of the PMI header of the next ethernet packet will be 0x01 instead of 0x00, to signal the error. To ensure that no byte-miscount occurs after such an error, the *pm3_rec_reset()* routine is invoked to wait until the next startbytes occur on the bus. If, for any reason the PMI reads more bytes than the phasemeter has provided or vice versa, a resulting block shift is avoided by this procedure. Block shift means, that when the PMI starts reading the first byte of a block, the received byte is in reality not the first byte of the data-block but rather located ahead or is part of the previous block.

To increase the speed, the EPP handshake is perfored by the *pm3_rec_recording()* routine itself and the timeout (Section 3.1.11) is only reset data-block-wise in contrast to the EPP driver, where the timeout is reset for each wait-loop. The *pm3_rec_timeout_flag* is checked only when the wait-condition of the wait-loops in the EPP handshake is fullfilled, which only occur, when the FIFO is empty. By this approach, the timeout does affect speed and latency only slightly.

The *pm3_rec_recording()* routine can return for two reasons: either the *pm3_rec_stop_flag* has been set by the *Stop* command or a timeout occured. After the recording routine returned, the *pm3_rec_stop()* routine is called to disable the recording mode in the phasemeter, re-enable message transmission (Section 3.1.10) and several other things.

Because that it is very time consuming to perform the EPP handshake by software, it were gainful to shift the handshake procedure to an additional hardware, either an appropriate controller chip like *Exar ST78C36*[24] or a FPGA. But since the PMI reaches a sufficient data rate, this feature is not intended. Rather, if a higher transfer rate is desired, the EPP interface should be avoided to read the data directly from the phasemeter's FIFO. For that, little changes in the phasemeter are necessary and a new driver has to be written for the PMI to replace the EPP driver.

As already mentioned, the recording operation can be stopped by the *Stop* command or, of course, by sending the *Reset* command. The packet size depends on the phasemeter configuration in general and in particular

on the selected number of channels which should be covered. According to the following equation, the block size range is 26 B to 444 B (the phasemeter supports up to 20 channels). One UDP packet contains one data block, covering all selected channels.

$$BlockSize = ChannelsBypes \cdot PerChannel + StartBytes$$

BytesPerChannel: 22
StartBytes: 4 (should be 0xff each)

### 3.1.4 EPP Driver

*Source files: epp.c, epp.h*
The EPP driver provides low level read and write functions, as specified in the EPP specification *IEEE1284* [18]. The EPP protocol defines four operations or handshakes, which are used by the PM3 driver and which are additionally directly accessable by PMI commands:

- Read Data
- Write Data
- Read Address
- Write Address

The driver is resposible for the EPP handshake and associated timeouts. Additionally, initialization and reset functions are provided. The data transmission is done bytewise. An extra handshake must be performed for every byte, which is quite time consuming, mainly if it is done by software like in this case.

In principle, it is possible to control and use the phasemeter only with this four commands, but it would be too slow, because that every byte must be requested and seperately sent in an extra UDP-packet. To improve the transfer rate and also to ease software development on the computer system, the PMI provides higher level or phasemeter specific functions through the PM3 driver (Section 3.1.3).

Due to the implemented timeouts, the excecution of a handshake takes more time than can be tolerated in the recording mode. For this reason, the recording routine does not use this driver to perform the handshakes but rather doing it itself with a faster timeout method (Section 3.1.3.3).

### 3.1.5 RS232 Driver

*Source files: rs232.c, rs232.h*
This driver provides a function to transmit character-strings via RS232. It is used as an additional interface to provide error and status messages. The driver lacks an initialization routine, because the initialization is done by the *AT91Bootstrap framework* (Section 15). The RS232 protocol configuration is shown in Appendix D.

### 3.1.6 Memory Management Unit (MMU) and Cache Configuration

*Source files: mmu.c, mmu.h, regions.h*
The PMI software with its code and data regions is located in SDRAM. As already mentioned in Section 3.2.1, the internal SRAM size of altogether 8 kB is too small for the PMI software, so that the speed advantage of this memory region cannot be used. But, in any case, accessing memory exept for the processor registers takes additional time which amount depends on how the memory is coupled to the processor[10].

To improve the situation and in particular to decrease the delay when accessing main memory, caches are implemented, which are of tightly coupled and very fast memory. The *AT91SAM9260* architecture is of a harvard style, which involves a seperate data and instruction cache. The instruction cache can be simply enabled and does not need any additional configuration. In contrast, the data cache depends on the MMU and virtual memory. The MMU and cache functionality is again provided by the *CP15* coprocessor[11].

The software module to configure the MMU and caches is taken and adapted from *ARM System Developer's Guide*[9, 10]. Only the inline assembly must have been modified for the *GNU GCC* compiler[31]. For more information about *GCC inline assembly*, refer to *ARM GCC Inline Assembler Cookbook*[42].

The PMI is configured to use both, instruction and data cache as well as the write buffer[17, 11]. The SDRAM memory region is splitted into five parts, which are listed in Table 1. It is Important to note, that a cache which covers data which again is additionally accessed by DMA, like it is the case for the EMAC buffers and descriptors, may need to be cleaned[11, 10] before DMA access takes place. In case of the PMI, only the TX buffer is cached to avoid cleaning. There, the cache yields to a better performance, because that the UDP and IP checksum routines have to access most of the data of to be sent packets.

The book *ARM System Developer's Guide*[10] give a good introduction

| Region name | Description | Size | Cached | Buffered |
|---|---|---|---|---|
| System | Code and data | 4 MB | write back | yes |
| Page Table | Refer to [10] | 1 MB | write through | yes |
| EMAC_cb | EMAC descriptors and RX buffer | 1 MB | no | no |
| EMAC_WT | EMAC TX buffer | 15 MB | write through | yes |
| Stack | Stack region for all processor modes | 43 MB | write back | yes |

Table 1: Memory regions of main memory (SDRAM)

in cache and MMU technologie and discusses the subject on the basis of two common *ARM* cores. The *AT91SAM9260* contains an *ARM9EJ-S* core which is similar to *ARM920T* which again is discussed in the book. For technical details, refer to the technical reference manual of the *ARM9EJ-S*[11].

### 3.1.7 Reset

*Source files: reset.c, reset.h*
As outlined in Figure 7, the reset button on the microcontroller board does not directly trigger a hardware- but a software-reset. The hence invoked interrupt handler *PMI_reset_ih()*, performs the hardware reset finally. The reason for this procedure is, that a reset may be triggered by any routine by having always the same reset-procedure. Before the hardware reset is triggered, terminition code is executed to reset the prescaler value of the *Real Time Timer* (RTT) and to send all queued messages, including a reset-message, via RS232. The reason to reset the prescaler value of the RTT is, that if the prescaler is configured with another value than zero, the following boot will fail. If the reset came due to the *Reset* command, the command will also be acknowledged by the *eth_rx_process()* function.

To perform the hardware reset finally, the functionality of the *Reset Controller* (RSTC)[17] is used to assert the NRST line of the processor[11, 17] by software and in particular by a corresponding write to a register.

### 3.1.8 Command Queue

*Source files: queue.c, queue.h*
The command queue constists of a FIFO (First In - First Out) memory to store the command code and additional information or data which come with the command. A queue-entry can store up to 4 B of data. The *queue_command()* function is to store a new command and invoked by the ethernet receive

module (Figure 12). The *get_ command()* function returns the oldest queue-entry and is run by the main loop (Figure 6).

### 3.1.9 Execute Command

*Source files: command.c, command.h*
This routine expects a command-queue-entry pointer as parameter and invokes the corresponding subroutine which processes the command. The routine consists of a branching which covers all available commands of the PMI, except the priority command *Reset*. If the command processing needs only a few lines of code, these are directly inbuild in the branching.

### 3.1.10 Message Handler

*Source files: message.c, message.h*
The message handler provides a function for the PMI to generate messages. The messages are queued in a FIFO with two independent outputs. This feature is needed, since the messages are transmitted frequently via RS232 and can be additionally accessed by the *Get Messages* command via ethernet.

The routine, which sends the queued messages via RS232, is invoked by a timer interrupt with a frequency of 4 Hz (Section 3.1.11). In the recording mode, the timer interrupt is disabled, because that the RS232 message transmission is relative time consuming. The other possibility to get the messages is, as already mentioned, to use the *Get Messages* command, on which the PMI sends all messages since the last access via ethernet, one message-string per UDP packet.

An important feature of the message handler is, that a timestamp is attached at the beginning of every message. The displayed time is the elapsed time since the last reset. The timestamp has the following format:

*hh:mm:ss>*

The function to generate a message is of a reentrant design. All interrupts are disabled when shared variables are accessed, to prevent data corruption. That makes it possible to invoke the *message()* routine more than once, for example from both, *task mode* and *exception mode* (interrupt mode) (Section 3.1).

### 3.1.11 Time

*Source files: time.c, time.h*
Several timing functions are centralized in this module, which are listed below. The timer initialization routines as well as control and status routines are placed here.

- Delay
- Timeout
- Timer interrupt
- Real-time clock initialization

The delay function *delay_ ms()* can be called with a time period in milliseconds as a parameter and it returns after the time has elapsed. To set up a timeout, the *timer_ set_ timeout()* function can be called, also with a time period in milliseconds as parameter. In contrast to the *delay_ ms()* function, the function returns immediately after setting up a timer. The task, which wants to use the timeout, has to poll the *timer_ compare* macro, which returns a flag which again signals whether the defined time has elapsed. The maximum delay or timeout period is 2047 ms. The *timer interrupt* function is used to invoke the RS232 message transmit function (Section 3.1.10). The timer is configured to generate interrupts with a frequency of 4 Hz. The interrupt of the timer can be enabled and disabled, which is used by the recording module, to disable the time consuming message transmission.

All of those three functions make use of one of the *Timer Counter* (TC) of the MCU[17]. The timers are configured to be connected to the *slow clock*, which has a frequency of 32.768 kHz. Hence, eight timer timer ticks occur within a millisecond in which the timer value is incremented eight times. Internally, the delay and timeout functions read the current value of the 16 bit *counter register*[17] of the used TC and add the desired number of milliseconds, multiplied by eight. The calculated value can now be set in the *alarm register*[17], which value is compared with the timer value on every increment. After the timer value equals the value in the *alarm register*, the alarm flag in the *status register* is set. This flag is polled by the delay and timeout function as well[7].

The module also provides a function to initialize the real-time clock, which is used to attach timestamps to messages (Section 3.1.10). The time, which can be read from the realtime clock, is the time in seconds since the last reset.

[7] **Timer Reset** The other possibility were to reset the timer value register to zero and to set the time period corresponding amount of timer ticks to the alarm register. But in practice, this solution will not work. After the timer reset is performed, the timer needs some time to set the value register to zero. Unfortunately, the time is not mentioned in the MCU data sheet, but you have to wait for it before the alarm flag can be cleared and afterwards polled. In the time between the reset is performed and it takes effect, the alarm flag may incorrectly signal that the time has elapsed. And waiting for that the reset takes effect will increase system latency.

### 3.1.12 Interrupt Handler

*Source files: interrupt.c, interrupt.h*
The PMI uses several interrupt handlers for the different interrupt sources. The handlers are written in C, but they are invoked by a piece of assembly code, to save the processor and register state to the stack. The *interrupt.c* file contains mostly only the first part of a handler, which task it is to examine the interrupt source, for which the interrupt has been triggered. For example, when an EMAC interrupt occurs, the handler first has to look from which part of the EMAC the interrupt is generated and then to run the specific handler.

### 3.1.13 Exception Handler

*Source files: crt0.S, exception.c, exception.h*
The *ARM926EJ-S* architecture provides several processor exception modes like *Data Abort*, *Prefetch Abort* and *Undefined Instruction*. In addition, there are two interrupt modes, which are not used by the PMI: *Fast Interrupt Request* (FIQ) and *Software Interrupt* (SWI). If one of these exceptions occurs, a meaningful message is generated and printed immediately via *RS232* (Section 3.1.10). The PMI-program execution is not affected by the exception handler but probably by the source of the exception.

### 3.1.14 Assembly Routines

*Source files: asm.S, asm.h, asm_ macro.c, asm_ macro.h*
The *asm.S* file contains a few assembly routines to invoke interrupt handlers as described in Section 3.1.12. The *asm_ macro.c*-file contains functions to enable and disable the IRQ-line of the processor. The *irq_ off()* routine returns a flag which signals whether the interrupt was already disabled or has been disabled by the routine. If the interrupt was already disabled, the CPU is in IRQ mode and the interrupt must not be enabled again.

## 3.2 Boot Sequence

To run the PMI software, two bootloaders are necessary. The general boot procedure is shown in Figure 14. The first loader is the embedded boot program[17], which is described in Section 3.2.1. After a reset or powering up, it loads the *AT91Bootstrap framework*[14] image from *DataFlash* at address 0x0 to internal SRAM and runs it.

Supposed, the embedded boot loader cannot find a valid image or the *DataFlash* unit is disabled, it searches additionally in NAND-flash. NAND-

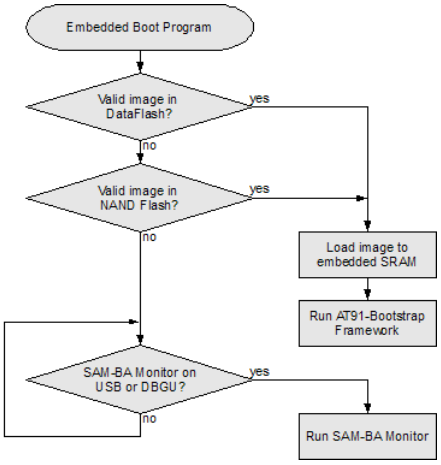



Figure 14: Embedded boot loader

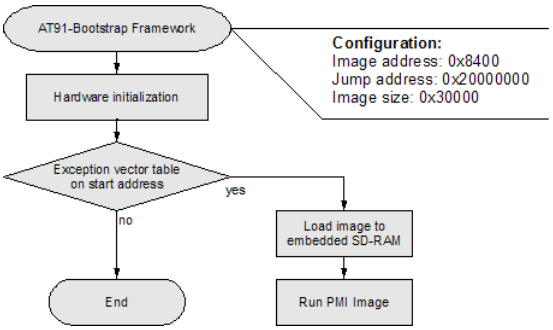



Figure 15: AT91Bootstrap Framework

flash is not used by the PMI and it might be problematic to use for booting, because of a MCU errata (refer to the MCU manual[17] or the board manual[48] for further information). If also the boot from NAND-flash fails, the MCU starts an embedded program and waits for a connection with the Atmel In-System-Programmer (ISP) *SAM-BA*[15]. This tool enables to load a program-image to *DataFlash*, for example.

The second bootloader is an adapted version of the *AT91Bootstrap framework* by Olimex. The function is outlined in Figure 15. After hardware initialization, the framework copies the actual PMI image from *DataFlash* at address 0x8400 to SDRAM at address 0x20000000 and runs it.

### 3.2.1 Embedded Boot Program

As already mentioned above, the embedded boot program searches for an image in *DataFlash* and NAND flash. The sequence is shown in Figure

14. Both memory types are available on the board and can be controlled by two jumpers, *DF_E* (DataFlash Enable) and *NANDF_E* (NAND Flash Enable)[48]. Because that the PMI does not use NAND flash, this memory should always be disabled. To run the PMI, the jumper *DF_E* has to be closed.

The image which should be run must be stored at address 0x0 in *DataFlash*. Its size must not exceed 4 kB, according to the internal SRAM size. If a valid exception vector table is found and the image size does not exceed the 4 kB limit, it will be copied to SRAM. Afterwards, a remap is performed and the program counter is set to 0x0. The remapping function enables to boot from several memory devices which are mapped in the address space. The remap command maps the appropriate device to 0x0 (Appendix C). For more detailed information about the boot sequence, refer to the MCU manual[17].

### 3.2.2 AT91Bootstrap Framework

The *AT91Bootstrap framework*[14] first initializes essential hardware components like SDRAM. If a valid exception vector table can be found, the image will be copied to SDRAM. Afterwards, the framework branches to the first word (the first instruction) of the image.

In the case of the PMI, the framework is configured to load the PMI image from *DataFlash* at address 0x8400. The destination or jump address in SDRAM is 0x20000000. The maximum length of the image is 0x30000, but can be adjusted in the source code.

The main reason to use a second boot loader is that the image size of the PMI software is potentially bigger than 4 kB[8]. Another reason for using *AT91Bootstrap framework* is the hardware initialization feature, including SDRAM and clock initialization.

Because that the exception vector table of the PMI image is copied to SDRAM only, the vector table of the framework is still used. The new table has to be copied to SRAM at address 0x0 by the PMI startup code, which is described in Section 3.3.1.2.

Note, that the adapted framework from *Olimex* is developed with *codesourcery*[2] and the use of something else may cause an erroneous build.

[8]Note, that the image size is much smaller than the needed memory of the program. All uninitialized variables are not part of the image but only reserved. The image contains only the code and initialized data and may have a length of less than 4 kB.

## 3.3 Other Initialization

Before the PMI takes service, the following modules must be initialized:

- C-runtime (startup-code)
- Ethernet PHY
- EPP extension board
- Advanced Interrupt Controller (AIC)
- Reset Controller (RSTC)
- Peripheral clocks

### 3.3.1 C-Runtime

The *startup-code* or *C-runtime* is an essential piece of assembly code, since the microcontroller's stack pointers and the *Current Program Status Register* (CPSR)[11, 9, 35] are not memory mapped and thus inaccessable from C-code. C-programs make use of the stacks and they must be initialized before any C-code is executed. A good introduction to this issue gives the article *Building Bare-Metal ARM Systems with GNU*[47].

The following listing shows all tasks of the startup code:

- Stack pointer initialization.
- Copying of the exception vector table.
- Zero all uninitialized variables.
- Enable interrupts.
- Branch to the main routine.

**3.3.1.1 Stack Pointer Initialization** The *ARM9* core contains several processor modes with particular stack pointer registers. The stack pointers of all modi should be initialized, even though they are unused. Usually and also in this case, the stack pointers point to the top of the stack and grows downwards. The first pointer points to 0x24000000, which is actually out of the memory range of SDRAM. But because that the pointer always points to the last used word, the pointer is decremented before data is pushed onto the stack. After switching to the next processor mode, the pointer address

**Exception vector table:**

| Address | Event | Mode | PMI Instruction | Alternative Use |
|---|---|---|---|---|
| 0x00 | Reset | Supervisor | Branch to startup code | - |
| 0x04 | Undefined Instruction | Undefined | Branch to corresponding handler | - |
| 0x08 | Software Interrupt | Supervisor | Branch to corresponding handler | - |
| 0x0c | Prefetch Abort | Abort | Branch to corresponding handler | - |
| 0x10 | Data Abort | Abort | Branch to corresponding handler | - |
| 0x14 | Reserved | - | - | Image size (Section 14) |
| 0x18 | IRQ | IRQ | LDR PC, [PC, # -&F20] (Section 3.3.1.4) | - |
| 0x1c | FIQ | FIQ | Branch to corresponding handler | - |

Table 2: ARM9 exception vector table

of the previous mode is substracted by 0x1000 and afterwards written to the current stack pointer register. Thus, every stack has a size of 4 kB which is sufficient for this project.

**3.3.1.2 Exception Vector Table** If an exception like an interrupt request occurs, the CPU sets the *program counter[11, 9, 35]* to an address which corresponds with the exception source. The exeption vector table can be seen in Table 2.

After remap (Appendix B), triggered by the *embedded boot program* (Section 3.2.1), the first SRAM bank is mapped to address 0x0. The SDRAM memory is mapped to 0x20000000 which is again the address of the PMI image with the preceding exception vector table. The table must be copied to 0x0 in SRAM. This task is done by the startup code.

**3.3.1.3 Zero Uninitialized Variables** All uninitialized variables in C must contain the zero value. The memory area of these variables is not copied to the program image by the linker but rather reserved. The startup code zeroes this area.

**3.3.1.4 Interrupts** Both interrupt types, *Fast Interrupt Request* (FIQ) and *Interrupt Request* (IRQ) can be controlled by the *Current Program Status Register* (CPSR)[11, 9, 35] of the processor. The PMI uses only the IRQ. The reset state of both interrupt types is disabled, therefore, the IRQ must be enabled by resetting the corresponding bit in the CPSR, before it can be used.

For IRQ handling, the *Advanced Interrupt Controller* (AIC)[17] is used. It contains a register called *Interrupt Vector Register* (IVR), which has to be

read after an IRQ occured. By the IRQ, the program counter is immediately set to the corresponding entry in the exception vector table (address 0x18). To read the IVR, the following assembly instruction must be set in this entry:

LDR PC, [PC, # −&F20]

The IVR returns the value written in the *Source Vector Register* (SVR), corresponding to the IRQ source. For each of the enabled interrupt sources, the SVR must be first initialized with the address of the appropriate interrupt handler, which is done by the corresponding software modules in the initialization phase. Refer to Section 3.3.4 for further information about AIC initialization.

**3.3.1.5 Start of the Main Routine** Finally, the C-runtime is set up and C-code, namely the main routine, can be run by a branch instruction to the corresponding address.

### 3.3.2 Ethernet PHY

The PHY is described in Section 4.1.1. Here, the initialization of the PHY itself and the interface which connects the PHY with the EMAC is described.

**3.3.2.1 PHY (interface)** The EMAC of the MCU lacks the physical layer, abbreviated PHY. The PHY is implemented as an external chip[46] on the board. For the connection between PHY and EMAC, the *Media Independend Interface* (MII) is used. Additionally, both components are connected via a *Management Data I/O Interface* (MDIO) to transmit status and control information from and to the PHY. Both must be configured and set up through the EMAC user interface registers, described in the MCU manual[17].

The pins of the MII and MDIO interface are multiplexed with other peripheral functions, so the pins must be set up according to their peripheral function. One more pin is used for the IRQ line of the PHY, to detect a link status change. This pin must be configured as a *Parallel I/O* (PIO) input with enabled interrupt. Refer to the MCU manual[17] for further information about the PIO-controller setup.

**3.3.2.2 PHY (itself)** Some settings in the registers are preset and others are set according to the logical state of the corresponding pin after reset. These pins must be pulled up or down, but they are nevertheless still useable for data transmission afterwards. If these settings are satisfying, no further

set up is necessary. To change the configuration, the EMAC provides a register for the MDIO communication. The PMI changes the configuration.

Often, further PHY access is needed to get link information about speed and duplex mode, to set up the EMAC accordingly. But in case of the PMI, this is not necessary. The Interface is designed to work only in the 100 Mbit and full-duplex mode. All information can be set during the initialization phase and do not need to be changed.

The PHY is configured to generate an interrupt when the link status changes. To ascertain whether the link has been broken or established, the interrupt status register of the PHY is read. The register flags are cleared automatically by reading.

### 3.3.3 EPP Extension Board

The 14 EPP bus lines are connected to the MCU's *Parallel I/O Controller* (PIOC)[17]. The corresponding pins must be configured according to their function. To avoid glitches, caused by transmission line reflections, crosstalk or other interferences to be considered, the inbuild glitch filter is enabled for all input lines. For more information about the EPP extension board, refer to Section 2.3.

### 3.3.4 Advanced Interrupt Controller (AIC)

The AIC handles up to 32 interrupt sources. Each source can be enabled or disabled separately in the *Interrupt Enable/Disable Command Register* (IECR and IDCR). For each enabled source, the address of the corresponding interrupt handler must be written in the *Source Vector Register* (SVR). Besides, the source type (edge triggered or level sensitive) and the interrupt source priority must be configured in the *Source Mode Register* (SMR). When an interrupt occurs, the IVR contains the value of the SVR corresponding to the interrupt source. Refer to the MCU manual[17] for further information about the AIC.

### 3.3.5 Reset Controller (RSTC)

By default (after a reset), the NRST pin of the MCU[11, 17] asserts directly a core and peripheral interrupt. The *Reset Controller* (RSTC)[17] initialization changes this configuration in a way, that an interrupt instead of a reset is gererated when NRST is asserted. Refer to Section 3.1.7 for further information.

### 3.3.6 Peripheral Clocks

Several embedded peripheral devices need a clock signal, which can be controlled by the *Power Management Controller* (PMC). In order to save energy, the clocks of the peripherals can be switched on and off by a write to the corresponding register. In the case of the PMI, all used peripherals are switched permanently on. Refer to the MCU manual[17] for further information about the PMC.

## 3.4 Configuration

For changes of the PMI configuration, important settings are collected in the *config.h* file. The following changes can be done:

- PMI MAC address
- PMI and host IP address
- PMI and host port
- Timeouts
- Buffer and queue sizes
- Maximum transfer unit (MTU)
- Table fragment size
- Enable/disable sequence number in UDP payload (Section 4.3)

| | OSI Layer | Protocol |
|---|---|---|
| 4 | Transport Layer | User Datagram Protocol (UDP) |
| 3 | Network Layer | Internet Protocol (IP) |
| 2 | Data Link Layer | Ethernet II (als known as DIX) |
| 1 | Physical Layer | Ethernet |

Table 3: Protocol stack

**Protocol interlacing**

| Ethernet II header | IP header | UDP header | Data | Ethernet II checksum |
|---|---|---|---|---|
| 14 bytes | 20 bytes | 8 bytes | 46 - 1500 bytes | 4 bytes |

Table 4: Protocol nesting

# 4 Communication

Actually, ethernet is not realtime capable and thus not suitable for our purpose. The need of a realtime connection originates in the imperatively guaranteed time period in which a packet must be transferred. However, since we have only a point to point connection of two hosts and we are working in full duplex mode, we can regard the connection as realtime. Collisions, which are responsible for the uncertain transmission time, occur only with more than two hosts in one collision domain.

The purpose for choosing ethernet is that it provides a great performance and most today's computer systems, as well as sophisticated microcontroller boards, are equipped with at least one ethernet port.

## 4.1 Protocol Stack

For the communication via ethernet, one specific protocol stack is supported. According to the *OSI Reference Model*, the stack consists of the protocols listed in Table 3. The PMI communication is done upon UDP packets.

As can be seen from the table, the ethernet specification defines the physical layer as well as the data link layer[53]. Each of these protocols are packet oriented, whereat every packet or frame consists of a header and a payload-field. A frame of an overlying layer will be encapsulated by the frame of a subjacent layer. Table 4 shows a network packet as it is designated for the PMI communication.

The PMI network configuration at a glance can be seen in Appendix D. As mentioned there, several protocol information are cosidered to distinguish between valid packets and those, which are not part of the PMI communication.

### 4.1.1 Physical Layer (PHY)

The Physical Layer is implemented by a so called PHY-chip of the Micrel *KS8721BL*-type[46]. It is a 10BASE-T/100BASE-TX transreceiver with an automatic speed and duplex configuration and an auto-crossover functionality. Owing to this, it is not needed to use a cross-over cable for a direct connection between the PMI and the computer system.

### 4.1.2 Data Link Layer (EMAC)

The next layer is also, even if partly, covered by hardware. This hardware is embedded in the MCU and is called *Ethernet Media Access Control* (EMAC)[17]. Its registers are mapped into the memory space and hence directly accessable from C-code. Network data is automatically transmitted by the *Direct Memory Access* (DMA) -controller of the MCU to SDRAM.

The EMAC provides several functions of its layer. The checksum, which appends every ethernet frame, can automatically be calculated and appended. On the other hand, received packets with an invalid checksum will be rejected. Those packets which pass the filter, are checked by the MAC-address filter again. There, the destination MAC-address is compared with a set of addresses in specific EMAC registers. Only those packets, which addresses matches one of the entries, are copied to memory. The EMAC can also be configured to bypass broadcast messages and even to copy all frames, but both options are disabled. The network configuration of the PMI can be found in Appendix D.

### 4.1.3 Network and Transport Layer

Both upper layers are implemented by software. Refer to Section 3.1.2.2 for further information about the protocol implementation.

## 4.2 Size Boundaries and Fragmentation

Both, UDP and IP Header have a length field of 16 bit, containing the size of the total datagram in units of bytes. Accordingly, the maximum length of each datagram is theoretically delimited by 65535 B. Substracting the size of 8 B of the UDP header, the UDP payload can have a length of maximum 65527 B. But an UDP datagram of this size can of course not be stored in a single IP datagram, because it has also a header which length has to be substracted. However, The UDP datagram is encapsulated in the IP datagram and the IP datagram in the Ethernet II datagram. The maximun length of

the UDP payload is delimited by the maximum length of the Ethernet II datagram of 1518 B. Indeed, it were possible to use the fragmentation feature to encapsulate huge UDP frames in a fragmented IP frame, but fragmentation is not supported by the PMI. After all, substracting all the headers from the maximum Ethernet II datagram length, the maximum payload for one UDP packet is 1472 B.

## 4.3 UDP Packet-Loss and Detection

UDP packet-loss might occur when the buffer size of the computer system is too small[7]. One possibility to detect loss is to check the ID field of the IP packet, which value is incremented on each packet. Where it is not possible to access this field, the PMI can be configured to copy the 16 bit sequence number to the beginning of the UDP payload. To control this feature, the *CPYSEQNBR* label can be defined or undefined in the *config.h* file (Section 3.4).

Even if packet loss will decrease the control bandwidth, the buffer enlargement as desrcibed in *UDP Buffer Sizing*[7], will add insult to injury. In this case, it is better to lost some older packets as to have to process them before the newer ones, after a lot of packets are accumulated.

## 4.4 Status and Error Messaging

To report error and status information, the RS232 interface is used to provide messages in plain text with a time stamp (Section 3.1.10). The RS232 interface is operable since the *AT91Bootstrap framework* (Section 15) has initialized it in an early state. To receive the messages, any computer or device with RS232 interface can be connected. For example, if the PMI should be monitored via intranet, a small webserver box can be used.

Internally, all messages are first queued and then periodically transmitted with a frequency of 4 Hz (except for the recording mode, Section 3.1.10). Additionally, the messages are also accessable by the *Get Messages* command. In this case, the PMI sends all messages since the last access via ethernet, with one message per UDP packet (Section 3.1.10). Neither the RS232 nor the ethernet transmission do affect each other by clearing the queue.

| Code | Meaning | Description |
|---|---|---|
| 01 | Identify | Send information about the PMI |
| 02 | Phasemeter Reset | Reset phasemeter |
| 03 | Set RAM Address | Set the phasemeter to a specific address |
| 04 | Set RAM Data | Write a 16bit value to the phasemeter |
| 05 | Read RAM Data | Read a 16bit value from the phasemeter |
| 06 | Set Channels* | Set the channel range in the phasemeter |
| 07 | Set NFFT* | Set the NFFT value in the phasemeter |
| 08 | Set Table | Write a whole sin/cos table to the phasemeter |
| 09 | Read Table | Read a whole sin/cos table from the phasemeter |
| 10 | Set PIR | Set PIR value in the phasemeter |
| 11 | Start Recording | Start recording in the phasemeter and send the data |
| 12 | Stop Recording | Stop recording in the phasemeter |
| 13 | Get Messages | Send all queued messages and clear queue |
| 14 | Write Address | EPP low level access: write address byte |
| 15 | Write Data | EPP low level access: write data byte |
| 16 | Read Address | EPP low level access: read address byte |
| 17 | Read Data | EPP low level access: write address byte |
| 18 | Reset | Reset the PMI |

*The value is internally read back to the PMI and checked.

Table 5: Interface commands at a glance

# 5 Using the Phasemeter through the PMI

The PMI provides full phasemeter access on a convenient and effective way. All in all 18 functions are provided, which can be addressed by a corresponding command code. Table 5 shows all commands at a glance with their particular codes. A detailed list can be found in Appendix C.

Within the communication between the computer system and PMI, the computer system works as a client and the PMI as a server. If the computer sends a command, the PMI will process it and acknowledges the result. Depending on the command, the computer system has to append additional data or, when data was requested, the PMI will provide it or otherwise return an error.

The PMI itself does not need any configuration or data for its operation. After connecting the power supply and a few seconds of booting and initialization, it is ready as long the ethernet link is established and the phasemeter is connected to the EPP port. The use of the RS232 is optional and will anyway not affect the operation. The purpose of the RS232 interface is to provide status and error messages in plain text, as described in Section

3.1.10.

## 5.1 Phasemeter Initialization

Before any phase-measurement can take place, the phasemeter needs to be initialized. The channel range, the number of supporting points of the DFT[37, 39] and the sin/cos-table must be set into the phasemeter. To ensure that the initialization data, which comes from the computer system, is valid and correctly stored in the phasemeter, the data can be validated by reading back and comparing with the original data. In the case of the *Set Channel* and *Set Nfft* commands, the verification is done by the PMI, which compares the original and read back value internally. The table must be read back to the computer system to compare it there.

The size of the sin/cos-table depends on the DFT parameters and can exceed a size of 100 kB. It must be transmitted in UDP packets, according to the communication protocol, with a maximum fragment size of 1464 B. If the fragment size crosses this boundary, the IP packet must be fragmented, what is not supported by the PMI. The fragment size, when the PMI sends the table back, is defined to be 512 B.

The user is free to choose a convenient order for the three initialization steps. The only thing which must precede these steps is a *Phasemeter Reset*. The *Set PIR* command is implemented for phasemeter testing purpose with an extra hardware. Setting this variable has no effect on the phasemeter operation and is only provided for the sake of completeness.

## 5.2 Byte Order

The byte order is *little endian*. This is contrary to the network byte order *big endian*, which is actually always used for network data transmission. But in our case, all systems including the phasemeter are working with little endian byte order, so it will save CPU time since it is not necessary to convert every word once on each side.

## 5.3 Command Acknowledgement

It is recommended to wait for the acknowledgement of the PMI, before the next command is sent. Nevertheless, if the computer system sends several commands at one time, they all will be processed in the right order. Refer to Section 3.1.8 for more information about command queuing.

## 5.4 Recording Mode

After initialization is successfully done, the phase-measurement can be started by sending the *Start Recording* command. If the preceding initialization phase was successful, the PMI will acknowledge the command positively and afterwards start data recording. Each sent data block contains the results of the DFT of all selected channels. To keep the latency low, each block is immediately send in an UDP packet. For further information about the outgoing data, refer to Section 3.1.3.3. Speed and latency issues are discussed in Section 3.1.1. Please note, that it is not recommended to operate the phasemeter with PMI up to the maximum possible frequency or phase-data-rate. It is worth to leave a safety margin of at least 10 %, to ensure an accurate operation.

## 5.5 Phasemeter Modification for Latency Optimization

The present behaviour of the phasemeter's EPP regarding the *wait*-signal, is not as it was primarily. The hard- and software of the phasemeter have been slightly modified to achieve a lower latency of the data transfer between phasemeter and PMI. The modification relates to the EPP *read data* handshake, where a byte is read from the phasemeters FIFO.

Previously, it was the task of the device which is connected to the phasemeter, to prevent a buffer underrun in the FIFO of the phasemeter. The practice was to wait until a particular amount of data is stored in the FIFO, before reading a data block. To detect the level of the FIFO, the *almost-empty*-flag[54] of the FIFO was observed, which signals whether the FIFO contains more or less than 1023 B. Because that the amount of bytes is 1023 and much higher than the maximum length of a data block (444 B), a buffer underrun became impossible. But this led to a severe latency of around 2.5 data blocks by covering all 20 channels and much more with a smaller number of channels.

The EPP specification states, that when a byte is requested from by the host by performing a *data read* EPP-handshake, the peripheral has to assert its *wait*-signal when it is ready to provide the data. In the previous version of the phasemeter, the *wait*-line was asserted regardless whether there is data in the FIFO. The result was, that when data is requested althought the FIFO is empty, the read data was corrupt. Refer to *The Parallel Port Complete*[18] for detailed information about EPP handshakes.

## 5.6 Phasemeter Documentation

There are several documents and papers about the phasemeter. For information about the DFT, refer to *Gerhard Heinzel*[39, 37]. For information about the phasemeter's operation purpose, refer to *Gerhard Heinzel, Vinzenz Wand or Iouri Bykov*[57, 38, 20].

# 6 Compiling & Programming

## 6.1 Development Environment

The software development of the PMI is done upon *GNU* development tools, namely *GCC* and *binutils*. Several open source toolchains are available for *ARM9* targets. However, I recommend the use of the commercial *codesourcery g++ ARM EABI* toolchain[2], which is based on *GNU* tools. A lite version without the eclipse based IDE (Integrated Development Environment) is available for free. *Codesourcery g++* can be run under linux as well as windows.

The exact target name is *ARM926EJ-S*, for which the program must be built.

## 6.2 Compiling

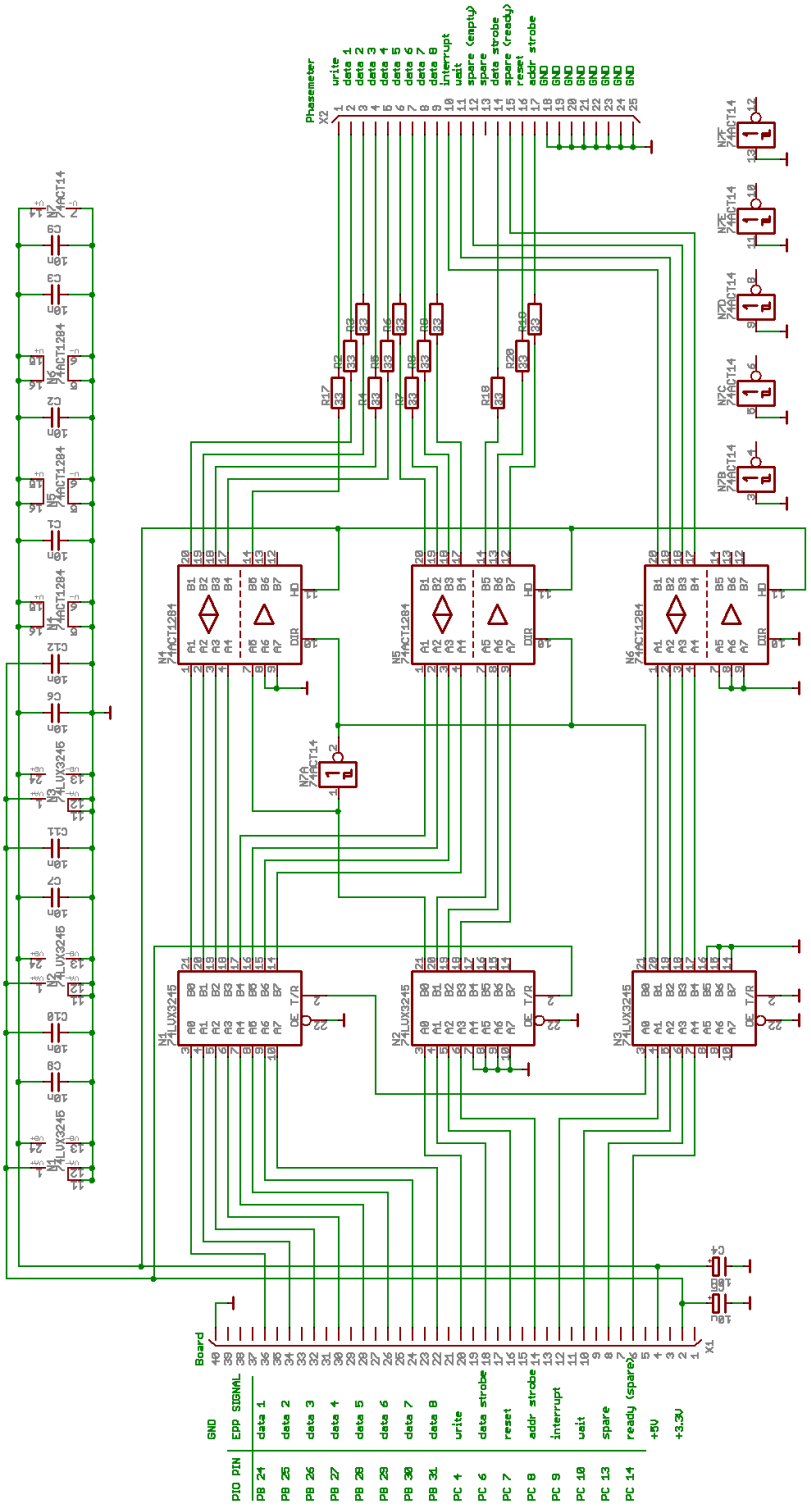
Compiling is straightforward: just type *make* in the source directory where also the file *Makefile* is located. The build is done automatically. The result is an image, which can directly be copied to the *DataFlash* memory of the microcontroller board.

After the installation of a toolchain, the makefile probably has to be modified. It contains a variable called *CROSS-COMPILE* which must be initialized with the name with which the toolchain is addressed.

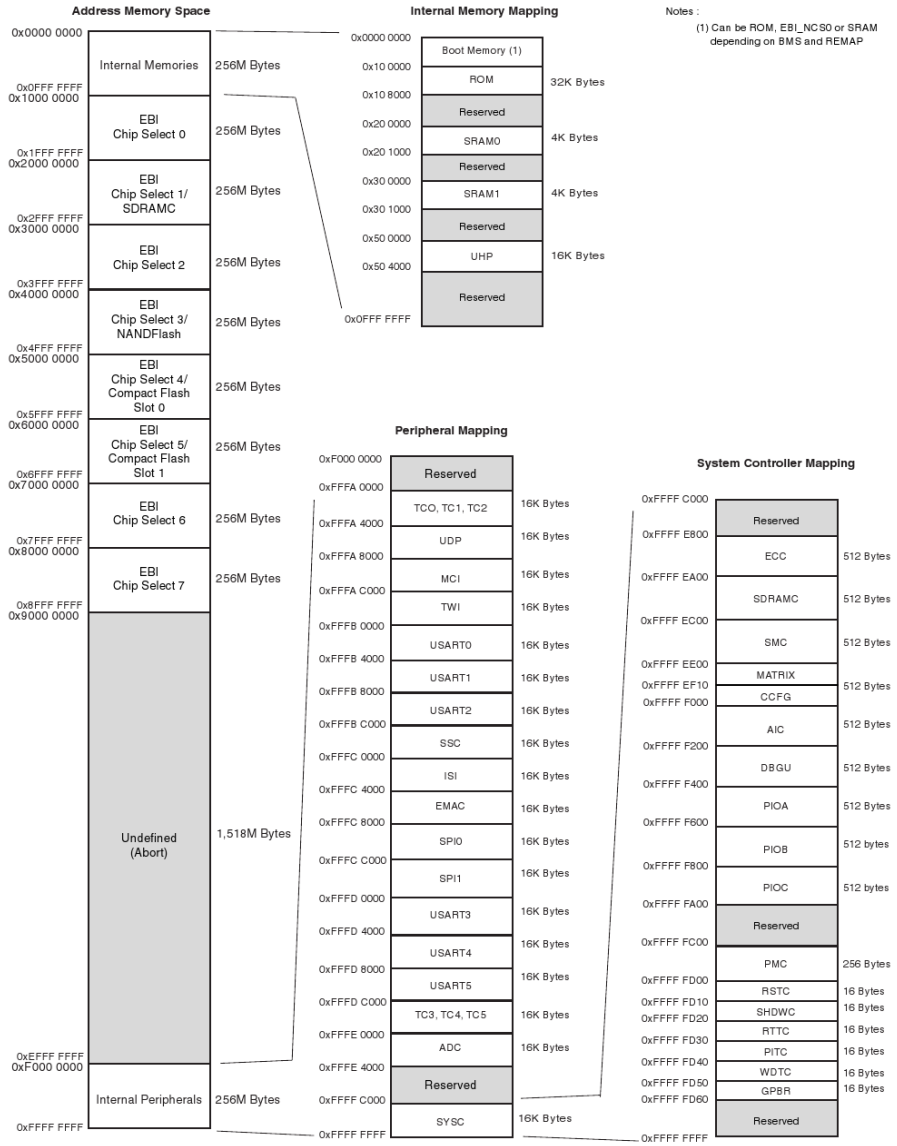
## 6.3 Flash Programming

To copy the binary to the *DataFlash* unit of the MCU-board, the In-System-Programmer (ISP) *SAM-BA* by *Atmel*[15] can be used. Its a convenient GUI-based program, available for windows and linux as well. The first step is to remove the DataFlash-enable *DF_E* and NAND-flash-enable *NANDF_E* jumper of the board (see Figure 2 in Section 2.2). A following reset will cause the MCU to run the embedded *SAM-BA monitor*, since no flash-memory could be detected (refer to the MCU manual[17] for further information). Now, the MCU-board can be connected with a computer system via USB and the *SAM-BA* ISP can be started. After starting the ISP, a board must be chosen by the user. The *Olimex SAM9-L9260* is similar to the *AT91SAM9260-EK*, which can be used. The *AT91Bootstrap framework* has to be copied to address 0x0 in *DataFlash* and the PMI-binary to address 0x8400. To start the PMI finally, a reset must be triggered and the the *DF_E*-jumper must be closed again.

# A Appendix Schematic of the EPP Extension Board



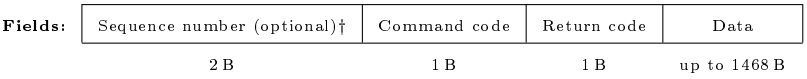
# B Appendix Memory Mapping



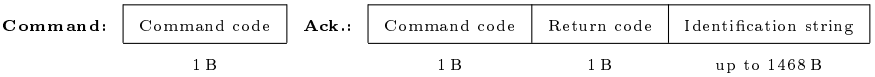

To understand the descriptions of the software modules, it is important to be aware of how the MCU uses its memory space. The microcontroller has a 32 bit address bus and can thus address $2^{32}$ B or 4 GB of memory. All attached memory devices as well as the user interfaces of the embedded peripherals, in terms of registers, are mapped into this memory space. For example, SDRAM is mapped to 0x20000000 and the second internal SRAM to 0x300000. Addresses are usually displayed with basis 16 and thus as hexadecimal values. The leading 0x of the mentioned addresses signals a hexadecimal notation. The whole memory map of the MCU can be found in Appendix B.

# C Appendix PMI Communication Protocol

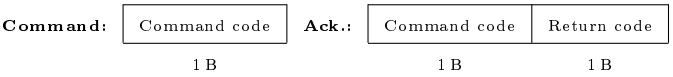
**General structure (for both directions)**



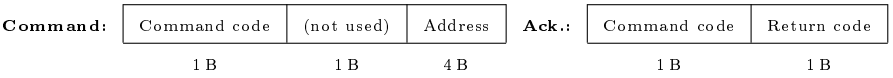

**Code: 01 Identify**



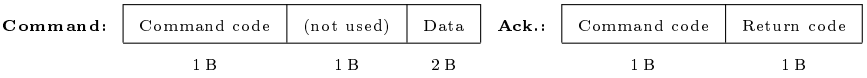

**Code: 02 Phasemeter Reset**



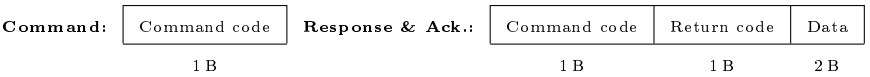

**Code: 03 Set RAM Address**



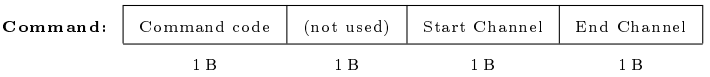

**Code: 04 Set RAM dData**

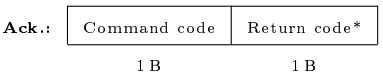



**Code: 05 Read RAM Data**

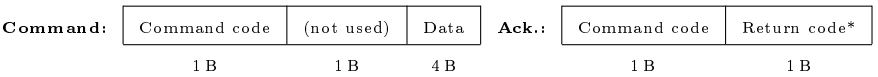



**Code: 06 Set cChannels**

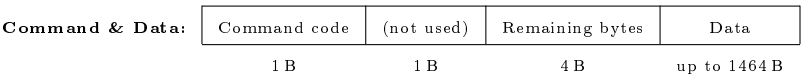


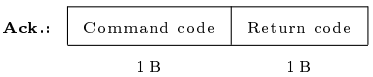




**Code: 07 Set NFFT**

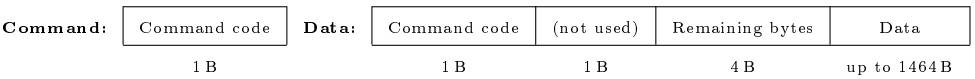


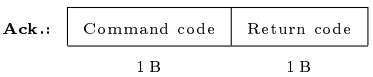

**Code: 08 Set Table**



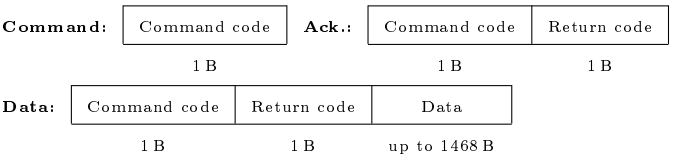

**Code: 09 Read Table**



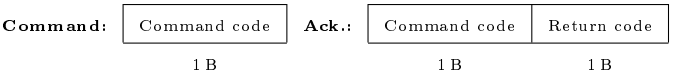



### Code: 10 Set PIR

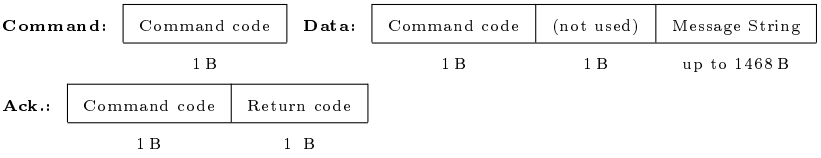
### Code: 11 Start Recording



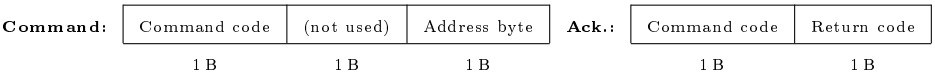

### Code: 12 Stop Recording



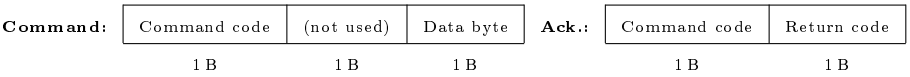

### Code: 13 Get Messages



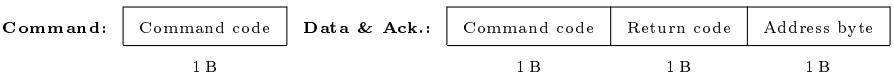

### Code: 14 Write Address (EPP - low level)



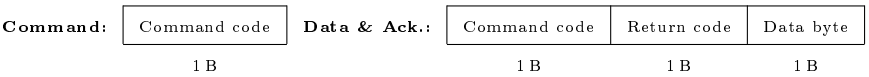

### Code: 15 Write Data (EPP - low evel)



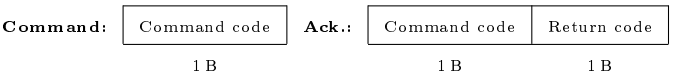

### Code: 16 Read Address (EPP - low level)




### Code: 17 Read Data (EPP - low level)




### Code: 18 PMI Reset




*The value is internally read back to the PMI and checked
†Refer to Section 4.3 for more Information.

### Return codes:

0: Successful

1: Error

# D Appendix PMI Configuration

### Ethernet

**PMI MAC address:** 00:01:29:D4:E2:5F*

**Host MAC address:** Set according to source MAC address of the first received and valid command packet

**Type:** 0x800 (IP)*

### Internet Protocol (IP)

**PMI IP address:** 192.168.7.2*

**Host IP address:** 192.168.7.1

**Protocol version:** 4

**ID field:** Incremented on each packet

**Fragmentation:** Tx: not used; RX: not accepted*

**Protocol:** 17 (UDP)*

**Checksum:** used*

**Optional header fields:** none

### User Datagram Protocol (UDP)

**PMI port:** 54321*

**Host port:** 54321

**Checksum:** used*

### RS232

**Baud rate:** 115200 bit/s

**Data bits:** 8

**Stop bits:** 1

**Parity:** None

**Handshake:** None

*These information are considered to distinguish valid packets.

# E Appendix Version History

## Ver. 1.0.0

First software release.

## Ver. 1.1.0

**Improvements** UDP and IP checksum algorithm improved. Thereby, the latency could be reduced to 15 µs for 20 channels.

## Ver. 1.1.1

**Bugfixes** All exceptions are now handled correctly by generating a meaningful message (Section 3.1.13). The program execution is not affected.

**TRM** The measurements in Section 3.1.1 are now consistent with the timings of the modified Phasemeter (Section 5.5).

## Ver. 1.1.2

**Bugfixes**

1. The *frame search* and *buffer clear* routine in *eth_rx_irq.c* can now handle also high transfer rates and will process all packets reliable.

2. The UDP checksum routine now calculates the correct checksum also for packets which are stored in buffers which are partly located at the end and partly at the beginning of the receive buffer array (with wrap around inbetween).

# References


[1] Atmel at91sam9260 emac-driver. http://www.atmel.com.

[2] Codesourcery g++ arm eabi toolchain. http://www.codesourcery.com.

[3] Compute 16-bit ones's complement sum. http://mathforum.org/library/drmath/view/54379.html.

[4] Geo600 project. http://geo600.aei.mpg.de/.

[5] Lwip - a leightweight tcp/ip-stack. http://savannah.nongnu.org/projects/lwip/.

[6] The red hat newlib c library. http://sourceware.org/newlib/libc.html.

[7] Udp buffer sizing. http://www.29west.com/docs/THPM/udp-buffer-sizing.html.

[8] Installing gcc. Linux Documentation Project, 2005.

[9] Chris Wright Andrew N. Sloss, Dominic Symes. Arm gcc inline assembler cookbook - code examples. http://www.elsevierdirect.com/companion.jsp?ISBN=9781558608740.

[10] Chris Wright Andrew N. Sloss, Dominic Symes. *ARM System Developer's Guide*. Elsevier, 2008.

[11] ARM Ltd. *ARM926EJ-S Technical Reference Manual*, 2008. Revision: r0p5.

[12] Atmel Corp. *Disabling Interrupts at Processor Level*, August 1998. Rev. 1156A-08/98.

[13] Atmel Corp. *AT91 Assembler Code Startup Sequence for C*, February 2006. Rev. 2644A-ATARM-06/02.

[14] Atmel Corp. *AT91Bootstrap framework*, October 2006. Version: V1.0.

[15] Atmel Corp. *SAM Boot Assistant (SAM-BA) User Guide*, October 2006. 6132C-ATARM.

[16] Atmel Corp. *GNU-Based Software Development on AT91SAM Microcontrollers Application Note*, March 2007. 6310A-ATARM.

[17] Atmel Corp. *AT91SAM9260 Datasheet*, July 2009. 6221I ATARM.

[18] Jan Axelson. *Parallel Port Complete*. Lakeview Research, 1997.

[19] Daniel Barlow. The linux gcc howto. Linux Documentation Project, 1999.

[20] Iouri Bykov. Phasemeter control and monitoring program. Sourcecode: pm3c.c and pm3d.c.

[21] Axel Sikora Christian Siemers. *Taschenbuch Digitaltechnik*. Fachbuchverlag Leipzig, 2003.

[22] Leroy Davis. Logic level translation. http://www.interfacebus.com/Design_Translation.html.

[23] Lewin A.R.W. Edwards. *Embedded Systems Design on a Shoestring*. Newnes, 2003.

[24] Exar Corp. *ST78C36/36A ECP/EPP parralel printer port with 16 byte FIFO*, August 2005. Rev. 5.0.2.

[25] Fairchild Semiconductor. *74ACT1284 IEEE1284 Transreceiver*, 2000.

[26] Fairchild Semiconductor. *IEEE1284 Interface Design Solutions*, 2000. AN-5010 Application note.

[27] Fairchild Semiconductor. *Simplified Intelligent Port Design Using the 74ACT1284*, 2000. AN-994.

[28] Fairchild Semiconductor. *74LVX3245 Data Sheet*, 2003. AN-994.

[29] Free Software Foundation. *The C Preprocessor*, 2007. Version 4.3.3.

[30] Free Software Foundation. *Using as - The GNU Assembler*, 2008. Version 2.19.51.

[31] Free Software Foundation. *Using the GNU Compiler Collection*, 2008. Version 4.3.3.

[32] Free Software Foundation. *The GNU Binary Utilities*, 2009. Version 2.19.51.

[33] Free Software Foundation. *The GNU linker*, 2009. Version 2.19.51.

[34] Klaus-Peter Köhn Friedrich Bollow, Matthias Homann. *C und C++ für Embedded Systems*. mitp, 2009.

[35] Steve Furber. *ARM-Rechnerarchitekturen für System-on-Chip-Design*. mitp, 2002.

[36] Jack Ganssle. Beginner's corner - reentrancy. http://www.ganssle.com/articles/begincornerent.htm.

[37] Gerhard Heinzel. Smart-2 ltp phasemeter. Draft - Version 0.3, June 2003.

[38] Gerhard Heinzel. The ltp interferometer and phasemeter. *Classical and Quantum Gravity*, February 2004.

[39] Gerhard Heinzel. Ltp interferometry frequency relationships. Draft - Version 1, October 2005.

[40] William Hohl. *ARM Assembly Language.* CRC Press, 2009.

[41] Edmund Jordan. *Embedded Systeme mit Linux programmieren.* Franzis, 2004.

[42] Harald Kipp. Arm gcc inline assembler cookbook. http://www.ethernut.de/en/documents/arm-inline-asm.html.

[43] Steve Maguire. *Writing Solid Code.* Microsoft Press, 1993.

[44] Peter Marwedel. *Embedded Systems Design.* Springer, 2006.

[45] Anthony Massa Michael Barr. *Programming Embedded Systems with C and GNU Development Tools.* O'Reilly, 2006.

[46] Micrel, Inc. *KS8721BL/SL Data Sheet*, 2005. Rev. 1.2.

[47] LLC Miro Samek, Quantum Leaps. Building bare-metal arm systems with gnu. *Embedded.com*, July/August 2007.

[48] Olimex Ltd. *SAM9-L9260 development board User Manual*, 2008.

[49] C. Partridge R. Braden, D. Borman. Computing the internet checksum. Technical report, Network Working Group, September 1988. RFC 1071.

[50] Peter R. Saulson. *Fundamentals of Interferometric Gravitational Wave Detectors.* World Scientific Publishing Co Pte Ltd, November 1994.

[51] Rob Savoye. Embed with gnu - porting the gnu tools to embedded systems. Cygnus Support, 1995.

[52] David E. Simon. *An Embedded Software Primer.* Adison Wesley, 1999.

[53] W. Richard Stevens. *TCP/IP, Der Klassiker, Protokollanalysen, Aufgaben und Lösungen.* Hüthig, 2008.

[54] Texas Instruments. *SN74V293 Datasheet*, February 2003. SCAS669D.

[55] Olaf Hagenbruch Thomas Beierlein. *Taschenbuch Mikroprozessortechnik*. Fachbuchverlag Leipzig, 2004.

[56] Krister Walfridsson. Aliasing, pointer casts and gcc 3.3. http://mail-index.netbsd.org/tech-kern/2003/08/11/0001.html.

[57] Dr.rer.nat. Vinzenz Wand. *Interferometry at Low Frequencys: Optical Phase Measurement for LISA and LISA Pathfinder*. PhD thesis, Gottfried Willhelm Leibniz Universität Hannover, 2007.

[58] Jürgen Wolf. *C von A bis Z*. Galileo Computing, 2006.